1. [24 points in all]

Recall this from the Map class you wrote; this is a Map from strings to ints:

```
typedef string KeyType;
typedef int ValueType;

class Map
{
  public:
    Map();
    bool insert(const KeyType& key, const ValueType& value);
    bool update(const KeyType& key, const ValueType& value);
    bool insertOrUpdate(const KeyType& key,
                                    const ValueType& value);
    bool erase(const KeyType& key);
    bool contains(const KeyType& key) const;
    bool get(const KeyType& key, ValueType& value) const;
    // other functions not shown
};
```

Consider this excerpt from a class representing a photo. For this problem, all we need to know is that every photo has a string indicating the subject of the photo and and int indicating the year the photo was taken:

```
class Photo
{
  public:
    Photo(string subj, int yr);
    string subject() const { return m_subject; }
    int year() const { return m_year; }
    // other functions not shown
  private:
    string m_subject;
    int m_year;
};
```

Assume that we have this global variable, accessible from anywhere in the program:

```
Map scm;   // short for subjectCountMap
```

This map is supposed to keep a count of the subject strings for every Photo object currently in existence in the program. If the first four Photo objects created were

```
Photo p1("Royce Hall", 2008);
Photo p2("bruin statue", 2012);
Photo p3("Royce Hall", 2008);
Photo p4("Royce Hall", 2011);
```

then scm would map ("bruin statue" to 1 and "Royce Hall" to 3. This, then, could be part of the implementation of the Photo constructor shown above:

```
Photo::Photo(string subj, int yr)
...
   int count = 0;
   scm.get(m_subject, count);  // count remains 0 if
                               // m_subject is not in map
   scm.insertOrUpdate(m_subject, count+1);
...
```

If we let the compiler generate the destructor, copy constructor, and assignment operator for the Photo class, the contents of scm would not always be accurate. (For example, consider this function:

```
void h()
{
    Photo p("inverted fountain", 1998);
}
```

If just before we called this function, scm contained no occurrences of "inverted fountain", then after returning from the function, scm would incorrectly contain one occurrence of that string; that's incorrect, because *after* returning from the function there are *no* existing Photo objects with that subject string, since the object p went away.) Therefore, you must declare and implement the destructor, copy constructor, and assignment operator.

In parts a, b, and c below, you may implement additional helper functions if you like. Write any helper function implementations in whichever of parts a, b, or c first uses it; you don't have to repeat its implementation if a later part also calls it. The scm map must never contain a string that maps to 0; if no Photo exists with a particular subject string, then that subject string should not be in the map at all.

a. [5 points]

Complete the implementation of the destructor for the Photo class:

```
Photo::~Photo()
{
   int cont = 0;
   scm.get(m_subject, count);
   if(count == 1)
      scm.erase(m_subject);
   else if( count != 0)
      scm.update(m_subject, count-1);
}
```

b. [5 points]

Implement the copy constructor for the Photo class:

```
Photo :: Photo (const Photo &copy)
{
    int count = 0;
    m_subject = copy.m_subject;
    m_year = copy.m_year;
    scm.get (m_subject, count);
    scm.insertOrUpdate (m_subject, count+1);
}
```

5

c. [5 points]

Implement the assignment operator for the Photo class:

✓

```
Photo &Photo::operator=(const Photo &copy)
{
    int count=0;
    scm.get(m_subject, count);
    if(count == 1)
        scm.erase(m_subject);
    else if (count != 0)
        scm.update(m_subject, count-1);
    m_subject = copy.m_subject;
    m_year = copy.m_year;
    scm.get(copy.m_subject, count);
    scm.insertOrUpdate(copy.m_subject, count+1);
}  return *this
```

4

d. [5 points]

Consider the following program. Correct answers to parts a, b, and c will result in the assertions being true. Assuming those correct answers, the program produces four lines of output.

```
Map scm;

int howmany(const Map& m, string s)
{
   int ct = 0;
   m.get(s, ct);  // ct remains 0 if s is not a key in m
   return ct;
}

void f(Photo p)
{
   cout << "Line 2: " << howmany(scm, "David") << " "
                      << howmany(scm, "Carey") << endl;
}

void g()
{
   Photo p1("David", 2012);
   Photo p2("Carey", 1995);
   Photo p3(p2);
   if (howmany(scm, "David") == 1)
   {
      Photo p4("David", 2008);
      p2 = p4;
      Map checkScm;
      checkScm.insert(p1.subject());
      checkScm.insert(p2.subject());
      checkScm.insert(p3.subject());
      checkScm.insert(p4.subject());
      assert(howmany(scm, "David") == howmany(checkScm, "David"));
      assert(howmany(scm, "Carey") == howmany(checkScm, "Carey"));
   }
   cout << "Line 1: " << howmany(scm, "David") << " "
                      << howmany(scm, "Carey") << endl;
   f(p1);
   cout << "Line 3: " << howmany(scm, "David") << " "
                      << howmany(scm, "Carey") << endl;
}

int main()
{
   g();
   cout << "Line 4: " << howmany(scm, "David") << " "
                      << howmany(scm, "Carey") << endl;
}
```

On the next page, write the four lines of output this program produces.

Complete the four lines of output the program on the previous page produces:

Line 1: 2 1
Line 2: 3 1
Line 3: 2 1
Line 4: 0 0

5

e. [4 points]

Here is a declaration for a Magazine class:

```
class Magazine
{
  public:
    Magazine(string t, string i, int pg, int yr, string subj);
    // other functions not shown
  private:
    string m_title;
    string m_issue;
    Photo m_coverPhoto;
    int m_pages;
};
```

Every magazine has a title, an issue, a cover photo, and a number of pages. We can construct magazines like this:

```
Magazine m1("Nerd World", "Feb 2012", 128, 1995, "Carey");
Magazine m2("The Chic Geek", "May 2011", 64, 2011, "David");
```

The constructor takes the magazine's title, its issue, its number of pages, the year its cover photo was taken, and the subject of its cover photo.

4

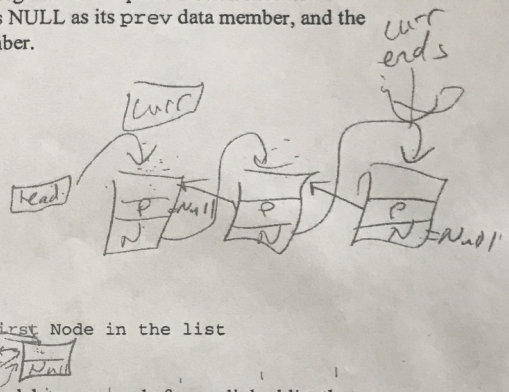
Write the implementation of the Magazine constructor below:

```
Magazine::Magazine(string t, string i, int pg, int yr, string subj)
{
  m_title = t;
  m_issue = i;
  m_pages = pg;
  m_coverPhoto(subj, yr);
}
```

2. [11 points]

Here is an excerpt from the definition of a doubly-linked list class. A LinkedList object represents a doubly-linked list of integers. The implementation uses no dummy node. The first node in the list has NULL as its `prev` data member, and the last node has NULL as its `next` data member.

```
class LinkedList
{
  public:
    ...
    void eraseSecondToLast();
  private:
    struct Node
    {
      int   value;
      Node* next;
      Node* prev;
    };
    Node* head;  // points to first Node in the list
};
```



The `eraseSecondToLast` function properly deletes one node from a linked list that has at least three nodes; it removes the one just before the last node in the list. You may assume that it will be called only for lists with at least three nodes. (In other words, the code you produce doesn't have to work for a list with two or fewer nodes.) The code **must** take the following form, with no additional lines and the blanks indicating code from the listed choices. Your options are limited to those shown.

```
void LinkedList::eraseSecondToLast()
{
    Node* curr = ____1____;
    while (____2____ != ____3____)
        ____4____ = ____5____;
    Node* temp = ____6____;
    ____7____ = ____8____;
    ____9____ = ____10____;
    delete ____11____;
}
```

- A `curr`
- B `curr->next`
- C `curr->next->next->next`
- D `curr->next->next->prev`
- E `curr->prev`
- F `curr->prev->next`
- G `curr->prev->prev`
- H `curr->prev->prev->next`
- I `head`
- J `head->next`
- K `NULL`
- L `temp`
- M `temp->next->next`

Write the letters corresponding to the filled-in code here; we'll look at these, not the code.

1	2	3	4	5	6	7	8	9	10	11
I	B	K	A	B	E	H	A	E	G	L

1. **[10 points]**

What is the output of the following program?

```
class Light
{
  public:
    Light() { cout << "L "; }
    ~Light() { cout << "~L "; }
};

class Clock
{
  public:
    Clock() { cout << "C "; }
    ~Clock() { cout << "~C "; }
};

class Signal
{
  public:
    Signal() { cout << "S "; }
    ~Signal() { cout << "~S "; }
  private:
    Light m_lights[3];
};

class TrafficSignal : public Signal
{
  public:
    TrafficSignal() { cout << "T "; }
    ~TrafficSignal() { cout << "~T "; }
  private:
    Clock m_clock;
};

int main()
{
  TrafficSignal ts;
  cout << endl;
  cout << "====" << endl;
  TrafficSignal* pts = &ts;
  cout << "====" << endl;
}
```

L L L S C T

====

~T ~C ~S ~L ~L ~L

2. [15 points in all]

Consider this excerpt from a rather ordinary class representing a Circle:

```
class Circle
{
  public:
    Circle(double x, double y, double r)
     : m_x(x), m_y(y), m_r(r)
    {}
    // other functions not shown
  private:
    double m_x;
    double m_y;
    double m_r;
};
```

A picture is a collection of circles. We choose to represent a picture as a dynamically allocated array of *pointers* to Circles. Here is an excerpt:

```
class Picture
{
  public:
    Picture(int capacity)
     : m_nItems(0), m_capacity(capacity)
    {
        m_items = new Circle*[capacity];
    }
    void addCircle(double r, double x, double y)
    {
        if (m_nItems < m_capacity)
        {
            m_items[m_nItems] = new Circle(x, y, r);
            m_nItems++;
        }
    }
    // other functions not shown
  private:
    Circle** m_items;
    int m_nItems;
    int m_capacity;
};
```

The first `m_nItems` elements of the `m_items` array contain pointers to dynamically allocated circles; the remaining elements have no particular value.

For parts a, b, and c below, you may implement additional helper functions if you like.

a. [5 points]

Complete the implementation of the destructor for the Picture class:

```
Picture::~Picture()
{
    for (int i=0; i< m_nItems ; i++)
        delete m_items[i];
    delete [] m_items;
}
```

b. [5 points]

Implement the copy constructor for the Picture class:

```
Picture:: Picture (const Picture& original)
{
    for (int i=0; i < m_nItems; i++)
        delete m_items[i];
    delete [] m_items;
    m_nItems = original. m_nItems;
    m_capacity = original. m_capacity;
    m_items = new Circle*[m_capacity];
    for(int x=0 ; x< m_nItems ; x++)
        m_items[x] = original.m_items[x];
}
```

c. [5 points]

Implement the assignment operator for the Picture class:

```
void Picture::swap(Picture& other)
{
    Circle** temp = m_items;
    m_items = other.m_items;
    other.m_items = temp;

    int nItems = m_nItems;
    m_nItems = other.m_nItems;
    other.m_nItems = nItems;

    int cap = m_capacity;
    m_capacity = other.m_capacity;
    other.m_capacity = cap;
}

Picture& Picture::operator=(const Picture& rhs)
{
    if(this != &rhs)
    {
        Picture temp(rhs);
        swap(temp);
    }
    return *this;
}
```

3. [30 points in all]

Consider the following linked list node structure:

```
struct Node
{
    Node(int v, Node* n) : value(v), next(n) {}
    int value;
    Node* next;
};
```

a. [15 points]

Write a function named `deleteNegative` that accepts a pointer to the head of a non-empty singly-linked list of Nodes. It is guaranteed the linked list will have at least one node, and that the first node of the linked list will ***never*** be negative. Your function should remove/free all nodes in the linked list that have a negative value. *You will receive a score of zero on this problem if the body of your deleteNegative function is more than 20 statements long.*

Hint: Don't forget to check your solution to make sure it works when there are multiple consecutive negative values in the linked list.

Here's how your function might be called:

```
int main()
{
    int a[6] = { 1, 2, -3, -1, -2, 3 };
    Node* h = NULL;
    for (int k = 5; k >= 0; k--)
        h = new Node(a[k], h);
    // h points to the first node in a list whose
    // nodes have the values 1 2 -3 -1 -2 3

    deleteNegative(h);
    // At this point, h points to the first node in a
    // list containing the values 1, 2, and 3.  The
    // other nodes containing -3, -1 and -2 should
    // have been unlinked from the list and deleted.
}
```

Write the `deleteNegative` function on the next page.

Write the deleteNegative function here.

```
void deleteNegative (Node* n)
{
    Node* p = n;
    Node* l = n->next;
    while (l != NULL)
    {
        if ((l->value) < 0)
        {
            Node* x = l;
            l = l->next;
            p->next = l;
            delete x;
            x = NULL;
        }
        else
        {
            l = l->next;
            p = p->next;
        }
    }
}
```

b. [15 points]

Write a **recursive** function named `equals`, which accepts two Node pointers, each pointing to a linked list, and returns a bool. The function returns true if the two linked lists have identical sequences of values (i.e., they have the same number of nodes, and corresponding nodes of each list have the same value) and false otherwise. Two empty lists are identical. *You will receive a score of zero on this problem if the body of your compare function is more than 12 statements long or if it contains any occurrence of the keywords* `while`, `for`, *or* `goto`.

Here's how your function might be called:

```
int main()
{
    Node* h1;
    Node* h2;
    … // Assume this omitted code sets h1 to point to
      // the first node in a list whose nodes have the
      // values 1 2 3, and sets h2 to point to another
      // such list.
    if (equals(h1, h2))  // true in this example
        cout << "Same!" << endl;
}
```

Write the `equals` function on the next page.

Write the `equals` function here.

```
bool equals (Node* n1, Node* n2)
{
    if ((n1 != NULL && n2 == NULL) || (n1 == NULL && n2 != NULL))
        return false;
    if (n1 == NULL && n2 == NULL)
        return true;
    if (n1->value == n2->value)
        return equals (n1->next, n2->next);
    return false;
}
```

Write the *first* digit of your UCLA student ID here: 6
Consider the `Person` structure and the two functions below:

```
struct Person {
    string name;
    int friends[3];  // 3 best friends
    bool asked;
};

void searchSocialNetwork(Person arr[], int start)
{
    queue<int> yetToAsk;
    yetToAsk.push(start);
    while ( ! yetToAsk.empty() )
    {
        int p = yetToAsk.front();
        yetToAsk.pop();
        if ( ! arr[p].asked )
        {
            arr[p].asked = true;
            cout << arr[p].name << endl;
            for (int k = 0; k < 3; k++)  // 3 friends
                yetToAsk.push(arr[p].friends[k]);
        }
    }
}

int main()
{
    Person people[10] = {
        /* 0 */ { "Lucy",   { 3, 5, 2 }, false },
        /* 1 */ { "Ricky",  { 7, 6, 5 }, false },
        /* 2 */ { "Fred",   { 0, 3, 8 }, false },
        /* 3 */ { "Ethel",  { 9, 0, 2 }, false },
        /* 4 */ { "Jerry",  { 6, 5, 8 }, false },
        /* 5 */ { "George", { 0, 1, 4 }, false },
        /* 6 */ { "Elaine", { 1, 9, 4 }, false },
        /* 7 */ { "Cosmo",  { 9, 8, 1 }, false },
        /* 8 */ { "Ralph",  { 4, 2, 7 }, false },
        /* 9 */ { "Ed",     { 3, 6, 7 }, false },
    };
    int s;
    cin >> s;  // Enter the first digit of your student ID number
    searchSocialNetwork(people, s);
}
```

If you enter the first digit of your student ID where indicated, what are the first six lines printed by the above program?

Elaine
Ricky
Ed
Jerry
Cosmo
George

15/15

5. [30 points in all]

Consider the following two classes:

```
class GasTank
{
  public:
    GasTank(double initGallons) : m_gallons(initGallons) {}
    void useGas(double gallons) { m_gallons -= gallons; }
    double getNumGallons() { return m_gallons; }
  private:
    double m_gallons;
};

class Battery
{
  public:
    Battery() : m_joules(10) {}
    void useEnergy(double joules) { m_joules -= joules;  }
    double getNumJoules() { return m_joules; }
  private:
    double m_joules;
};
```

For the first part of this problem, you are to write a Car class that is intended to be used as a base class. Here are the characteristics of a Car:

- A Car has a GasTank.
- When a Car is constructed, the user may specify how many gallons of gas a Car starts with in its GasTank. If this parameter is omitted during construction, a Car starts with 0 gallons of gas.
- A Car has a `drive` method that accepts a double specifying the number of miles to drive. The function should return the actual number of miles traveled as a double (which may be less than the parameter value if the Car runs out of gas during the trip). Cars can drive 10 miles per gallon of gas.
- A Car has a `cruiseControl` method that takes no arguments and when called, drives exactly 100 miles or as many miles as it can if there is not enough fuel to go 100 miles. It should return the number of miles actually traveled as a double.
- A Car has a `gallonsLeft` method that takes no arguments and returns a double indicating how many gallons are in its gas tank.
- You may declare a destructor, if appropriate.

You may assume (i.e., you do not have to check) that the doubles passed to the constructor and the `drive` method are nonnegative.

a. [15 points]

Declare and implement a `Car` class that provides the functionality above. Avoid needless redundancy in your implementation.

```
class Car
{
  public:
    Car(double initGallons = 0.0);
-2  virtual ~Car();
    virtual double drive(double miles);
    double cruiseControl();
    double gallonsLeft() const;
  private:
    GasTank* GT;
};

Car::Car(double initGallons)
{
  GT = new GasTank(initGallons);
}

Car::~Car()
{
  delete GT;
}

virtual double Car::drive(double miles)
{
  if (GT->getNumGallons() >= (miles/10))
  {
    GT->useGas(miles/10);
    return miles;
  }
  else
  {
    double numMiles = 10 * GT->getNumGallons();
    GT->useGas(GT->getNumGallons());
    return numMiles;
  }
}
```

```
double Car::cruiseControl()
{
    return drive(100.0);
}
double Car::gallonsLeft() const
{
    return GT->getNumGallons();
}
```

b. [15 points]

Next, define a class named `HybridCar`. A HybridCar is a kind of Car, but it also has multiple internal batteries. Define your HybridCar class with the following characteristics. (Do not redefine any member functions unless it is absolutely necessary!):

- The user must provide an int specifying how many batteries a HybridCar starts out with when it is constructed. A HybridCar's gas tank starts with 5 gallons of gas for each battery it has. So, for example, a HybridCar with 30 batteries would start out with 150 gallons of gas.
- Every HybridCar has a `drive` method which accepts a double specifying the number of miles to drive. The drive method should try to run entirely off the vehicle's batteries (using their power first); if all battery power runs out before the trip is complete, the car's gas should then be used. A HybridCar can travel 1 mile per Joule of electricity and 10 miles per gallon of gas. The function should return the actual number of miles traveled (which may be less than the parameter value if theHybridCar runs out of battery power and gas during the trip).
- A HybridCar may have a destructor, if necessary.

You may assume (i.e., you do not have to check) that the int passed to the constructor and the double passed to the `drive` method are nonnegative.

On the next page, declare and implement a HybridCar class that provides the required functionality. Avoid needless redundancy in your implementation.

Your class must be defined so that the following compiles, and when run, prints `Success`:

```
void getOutOfMyDreamsGetIntoMyCar(Car& c)
{
    c.cruiseControl();
}

int main()
{
    HybridCar myPrius(3);
    getOutOfMyDreamsGetIntoMyCar(myPrius);
    if (myPrius.gallonsLeft() == 8)  // not 5
        cout << "Success" << endl;
}
```

Write your declaration and implementation of `HybridCar` here. Avoid needless redundancy in your implementation.

9

```
class HybridCar: public Car
{
 public:
    HybridCar(int numBatteries);
    ~HybridCar();
    virtual double drive(double drive);
 private:
    Battery** batteries;
    int m_nB;
}
HybridCar::HybridCar(int numBatteries)
 : Car(5.0 * numBatteries), m_nB(numBatteries)
{
    batteries = new Battery*[m_nB];
    for(int i=0; i<m_nB; i++)
        batteries[i] = new Battery();
}
HybridCar::~HybridCar()
{
   for(int i=0, i<m_nB; i++)
      delete batteries[i];
   delete [] batteries;
}
virtual double HybridCar::drive(double miles)
{
      if(miles ≤ (10* m_nB))   // fraction? -2
      {
        for(int i= 0; i< miles/10; i++)
            batteries[i]-> useEnergy(10.0);
        batteries[static_cast<int>(miles/10)]-> useEnergy (miles % 10);
        return miles;
      }

      for(int i=0; i<m_nB; i++)
          batteries[i]->useEnergy(10.0);
```

```
if(miles ≤ gallonsLeft())
{
  return Car::drive(miles) + 10*m_nB;
}
else
{
    miles = miles - Car::drive(10*gallonsLeft());
    return miles;
}
}
```

some used

-2

3. [15 points]

Here is a non-recursive implementation of Problem 2's `fixNegatives` function that in addition, writes the data at each node it visits. (For this code and any changes you make to it for part b, assume any required headers have been included and that `using namespace std;` is present.

```
int fixNegatives(Node* p)
{
1       int count = 0;
2       queue<Node*> toDo;
3       toDo.push(p);
4       while ( ! toDo.empty())
        {
5           p = toDo.front();
6           toDo.pop();
7           if (p != nullptr)
            {
8               cout << " " << p->data; // write a number
9               if (p->data < 0)
                {
10                  p->data = 0;
11                  count++;
                }
12              for (int k = 0; k < 4; k++)
13                  toDo.push(p->child[k]);
            }
        }
14      return count;
}
```

a. If this function is called with a pointer to the node whose data value is –4 in the tree shown in Problem 2, what is written by all the executions of the indicated output statement? Circle the letter of your choice, and if it's K, write the output.

A. -4
B. -4 -1 5 -7 -3 -2 9 8 -6
C. -4 -1 5 -7
D. -4 -1 -2 9
E. -4 -1 -2 9 5 8 -6 -7 -3
F. -4 -1 -2 9 5 -7 -3 8 -6
G. -4 9 -6 8 -2 -1 5 -3 -7
H. -4 9 -2 -1
I. -4 9 -2 -1 -6 8 -3 5 -7
J. -4 9 -1 5 -6 8 -2 -3 -7
K. Something else. Write that something else here:

b. Indicate one-line replacements for no more than three of the numbered lines of the above function so that if it is called with a pointer to the node whose data is –4 in the Problem 2 tree, the output would be -4 9 -6 8 -2 -1 5 -3 -7 (If no replacements are necessary, then write "No replacements".) Replacement text must not use the name `fixNegatives` (so don't introduce recursion into this code).

Line number: 2 Replacement: stack<Node*> toDo;

Line number: 5 Replacement: p = toDo.top();

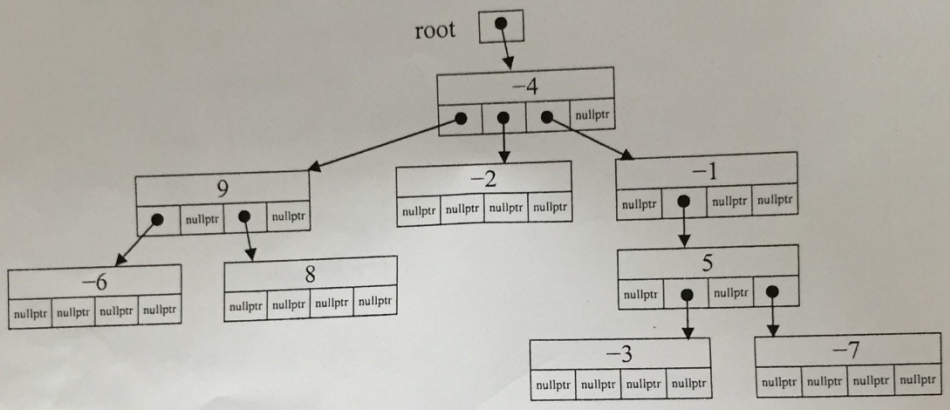
Line number: 12 Replacement: for (int k=3; k>=0; k--)

2. [15 points]

A 4-tree is a data structure that employs nodes defined as follows:

```
struct Node
{
    int data;
    Node* child[4];
};
```

Each element of the `child` array is either nullptr, denoting an empty subtree, or a pointer to a non-empty subtree. Here's a picture of one possible 4-tree:



The variable `root` is a `Node*` that points to a node with the data −4. That node's four children are the tree rooted at the node whose data is 9, the one-node tree whose data is −2, the tree rooted at the node whose data is −1, and an empty tree.

Write a **recursive** function named `fixNegatives` that takes as its one parameter a pointer to a node and replaces every negative data value with 0 in the tree rooted at that node, returning the number of values so replaced (which will be 0 if the tree is empty). For example, the statement

```
int n = fixNegatives(root);
```

would replace the six negative values in the tree shown above with 0, and set `n` to 6. If instead of that statement, we had executed the statement

```
int n = fixNegatives(root->child[2]);
```

with the tree shown above, the three negative values −1, −3, and −7 would be replaced by 0, and `n` would be set to 3. The statement

```
int m = fixNegatives(nullptr);
```

would set `m` to 0, since the empty tree has no negative data values to replace.

(continued on next page)

Your function must use recursion in a useful manner. It must not declare any variables with the keyword `static`, and must not use any global variables. If you declare any local variables, they must be of type `int` or `Node*`. The body of your function must contain no more than 15 statements (this is more than enough). **Violating these constraints will result in a score of zero for this problem.** You may use a loop if you wish, as long as your function uses recursion in a useful manner.

Write your function here:

```
int fixNegatives(Node* p)
{
    if (p == nullptr)
        return 0;
    int count = 0;
    if (p->data < 0)
    {
        p->data = 0;
        count++;
    }
    for (int i=0; i<4; i++)
        count += fixNegatives(p->child[i]);
    return count;
}
```

1. **[10 points]**

What is the output of the following program?

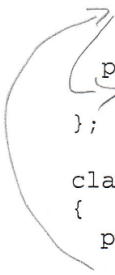
```
class Wing
{
  public:
    Wing() { cout << "W "; }
    ~Wing() { cout << "~W "; }
};

class MagnificentTail
{
  public:
    MagnificentTail() { cout << "M "; }
    ~MagnificentTail() { cout << "~M "; }
};

class Bird
{
  public:
    Bird() { cout << "B "; }
    ~Bird() { cout << "~B "; }
  private:
    Wing m_wings[2];
};

class Peacock : public Bird
{
  public:
    Peacock() { cout << "P "; }
    ~Peacock() { cout << "~P "; }
  private:
    MagnificentTail m_tail;
};

int main()
{
  Peacock p;
  cout << endl;
  cout << "====" << endl;
  Peacock* pp = &p;
  cout << "====" << endl;
}
```



W W B M P 5

= = = = 2

= = = =

~P ~M ~B ~W ~W 3

3. [35 points in all]

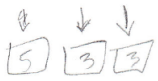
Consider the following class that implements a doubly-linked list of integers with no dummy node. The last node's `next` pointer is NULL; the first node's `prev` pointer is NULL. There is no tail pointer. When the list is empty, `head` is NULL.

```
class LinkedList
{
  public:
    ...
    int countAdjacentMatches() const;
    void eraseLast()
    {
        if (head != NULL)
            eraseLastAux(head);
    }
    void writeDiff(const LinkedList& other) const
    {
        wd(head, other.head);
    }
  private:
    struct Node
    {
      int value;
      Node* next;
      Node* prev;
    };
    Node* head;
    void eraseLastAux(Node* h);
    void wd(const Node* p1, const Node* p2) const;
};
```

a. [6 points]

The `countAdjacentMatches` member function counts how many nodes have a value that is equal to the value of the node that immediately follows it in the list. For example, if the LinkedList a contained nodes with the values 3 6 6 17 4 4 4 8 4 7, then the call `a.countAdjacentMatches()` returns 3, because it counted the first 6 and the first two 4s.

Write the `countAdjacentMatches` member function on the next page. *You will receive a score of zero on this problem if the body of your `countAdjacentMatches` member function is more than 15 statements long.*



[Write your `countAdjacentMatches` member function here.]

```
int LinkedList::countAdjacentMatches() const
{
 int count = 0;
 if(head == NULL) return count;
 for(Node* curr=head; curr->next != NULL; curr = curr->next)
     if(curr->value == curr->next->value)
         count++;
 return count;
}
```

b. [7 points]

The `eraseLast` member function removes the last node, if any, from the linked list. To help it do its work, it calls `eraseLastAux`, which removes the last node from the linked list, but is guaranteed to be called on a list with one or more nodes; it will always be passed a non-NULL pointer.

In the space below, write a **non-recursive** implementation of the `eraseLastAux` member function. *You will receive a score of zero on this problem if the body of your `eraseLastAux` member function is more than 15 statements long.*


```
void LinkedList::eraseLastAux(Node* h)
{
 Node* toErase = h;
 for(; toErase->next != NULL; toErase = toErase->next) ;
 if(toErase->prev == NULL)
     head = NULL;
 else
     toErase->prev->next = NULL;
 delete toErase;
}
```

c. [7 points]

Now write a **recursive** implementation of the `eraseLastAux` member function. *You will receive a score of zero on this problem if the body of your `eraseLastAux` member function is more than 15 statements long or if it contains any occurrence of the keywords `while`, `for`, or `goto`.*

5 4 3

```
void LinkedList::eraseLastAux(Node* h)
{
if(h->next == NULL)
    {
    if(h->prev == NULL)
       head = NULL;
    else
      h->prev->next = NULL;
    delete h;
    }
else
    eraseLastAux(h->next);
}
```

✓

p1 2 3 5 8 9 10
p2 3 4 5 6 7 8 9

d. [15 points]

A *strictly increasing list* is a list each of whose elements has a value that is less than the one that follows it. For example, 3 7 8 10 is a strictly increasing list, but 3 8 7 10 is not (8 is not less than 7), and 3 7 7 10 is not (7 is not less than 7). If `x` and `y` are LinkedLists whose nodes form two strictly increasing lists, calling `x.writeDiff(y)` writes out, one per line, all elements of `x` that are not in `y`. For example, if `x` contains 2 3 5 8 9 and `y` contains 3 5 6 7 8 10, then `x.writeDiff(y)` would write, one per line, the values 2 and 9.

The member function `writeDiff` calls a helper function `wd`, which accepts two Node pointers; if each points to a (possibly empty) strictly increasing linked list of Nodes, it writes out the values, one per line, that are in the first list but not the second.

In the space below, write a **recursive** implementation of the `wd` member function. You should assume that each of the lists is a (possibly empty) strictly increasing list. *You will receive a score of zero on this problem if the body of your `wd` member function is more than 20 statements long or if it contains any occurrence of the keywords `while`, `for`, or `goto`.*

```
void LinkedList::wd(const Node* p1, const Node* p2) const
{
 if(p1 == NULL) return;
 if(p2 == NULL)
   {
   cout << p1->value << endl;
   wd(p1->next, p2);
   return;
   }
 if(p1->value < p2->value)
   {
   cout << p1->value << endl;
   wd(p1->next, p2);
   }
 else
   {
   wd(p1, p2->next);
   }
}
```

insert here sorry!

```
if(p1->value == p2->value)
  {
  wd(p1->next, p2->next)
  return;
  }
```

5. [25 points in all]

Consider this attempt to write two classes that define an alarm clock type and a cuckoo clock type, and two functions that use these types:

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

     typedef int Time;

     class AlarmClock
     {
       public:
01:      AlarmClock(Time current) { m_time = current; }
02:      void setAlarm(Time alarm){ m_alarm = alarm; }
03:      void tick() {
04:          cout << "tick\n";
05:          if (++m_time == m_alarm) playAlarm();
06:      }
07:      void playAlarm()       { cout << "Buzz! Buzz!\n"; }
08:    private:
09:      Time m_time;
10:      Time m_alarm;
     };
11:  class CuckooClock : public AlarmClock
12:  {
13:    public:
14:      CuckooClock(Time t)     { m_time = t; }
15:      virtual ~CuckooClock() { cout << "Squawk!\n"; }
16:      void setAlarm(Time t) {
17:          cout << "Chirp!\n";
18:          m_alarm = t;
19:      }
20:      void playAlarm()  { cout << "Cuckoo! Cuckoo!\n"; }
     };

     void timePasses(AlarmClock& ac)
     {
          ac->setAlarm(503);
          ac->tick();
          ac->tick();
     }

     int main()
     {
         CuckooClock cc(501);
         timePasses(cc);
     }
```

When we run this program, we would like the output it produces to be

```
Chirp!
tick
tick
Cuckoo! Cuckoo!
Squawk!
```

a. [10 points]

Make the **minimal** changes **necessary** to correct the two classes so that the program builds successfully and produces the desired output above. You must in addition introduce const where appropriate. There are several restrictions:

1. You may modify only the lines in the program that we've indicated with numbers (so you must not modify the `timePasses` or `main` functions).
2. You must not add, remove, or change any statements that use `cout` (so you must not change the text of any quoted string literals).
3. You must not add or remove any member functions in either class.
4. You must not add or remove any data members in either class.
5. You must not add or remove the words `public`, `private`, or `protected`.

Notice that a correct solution may not necessarily use good C++ style, but it produces the required output.

List the line numbers and changes below that implement the correct solution, in a style like this example, where we suppose there were a line 21 with a function incorrectly declared:

`21: void foo()` ➔ `int* foo()`

Write your answer in the space below.

14: CuckooClock(Time t) : AlarmClock(t) { } +2

07: void playAlarm -> virtual void playAlarm +2

18: m_alarm = t; -> AlarmClock::setAlarm(t); +2

02: virtual void setAlarm(Time alarm) { m_alarm=alarm; }

+2

8

b. [3 points]

In part c, we'll define a new class named VoiceAlarm. A VoiceAlarm is a kind of AlarmClock that meets the following requirements:

1. When you create a VoiceAlarm, you must specify a string that will be the message played when it's time to play the alarm.
2. All VoiceAlarms start out with a current time of 1200.
3. Your VoiceAlarm constructor must use `new` to dynamically allocate a C++ string for the message and use a pointer data member to point to that dynamically allocated string.
4. When a VoiceAlarm's alarm is to be played, it must write the saved message instead of buzzing.
5. A VoiceAlarm must not leak memory.

Here's how a VoiceAlarm might be used:

```
int main()
{
    AlarmClock* a;
    …
    a = new VoiceAlarm("Wake up, slacker!\n");
    …
    delete a;
}
```

If you must make any changes to the AlarmClock base class to ensure your VoiceAlarm class works properly, state the changes in one or two sentences below; if you need not make any changes, write *No changes*.

AlarmClock must have the function
virtual ~AlarmClock() {}

3

c. [12 points]

On the next page, define and implement the VoiceAlarm class. Avoid needless redundancy; for example, the VoiceAlarm class must not declare a data member of type Time or int. (Also, for this problem you do not have to concern yourself with a copy constructor or an assignment operator.)

[Write your definition and implementation of VoiceAlarm here.]

```
struct VoiceAlarm : public AlarmClock
{
   VoiceAlarm(const string & alarm) : AlarmClock(1200)
      {
      m_alarmMes = mew string(alarm); //uses string copy constructor
      }
   ~VoiceAlarm() { delete m_alarmMes; }
   virtual void playAlarm() { cout << *m_alarmMes; }
private:
   string* m_alarmMes;
};
```

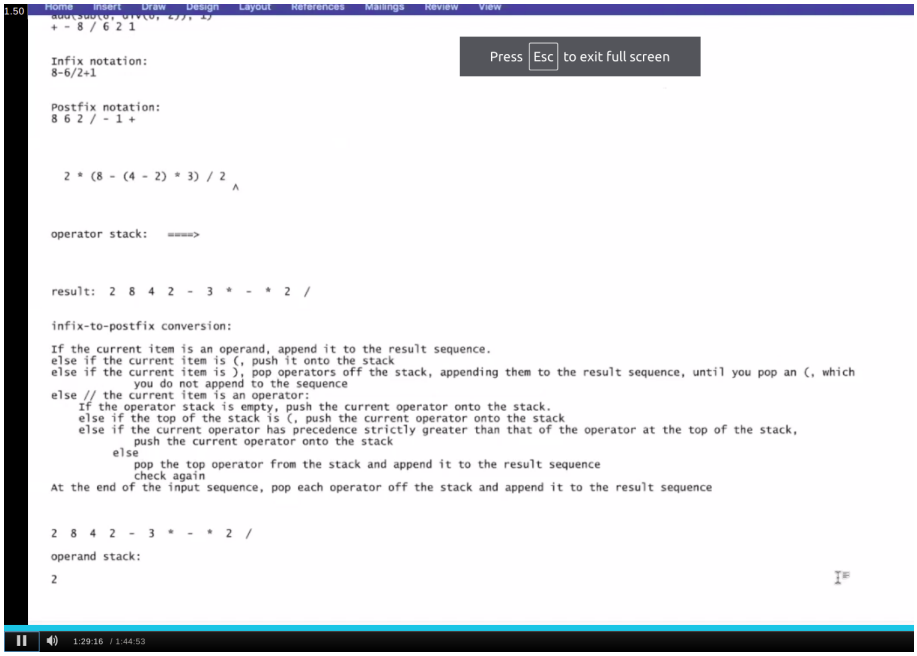
Const -1

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)


Home Insert Draw Design Layout References Mailings Review View
+ - 8 / 6 2 1
Press Esc to exit full screen
Infix notation:
8-6/2+1
Postfix notation:
8 6 2 / - 1 +
2 * (8 - (4 - 2) * 3) / 2
operator stack: ====>
result: 2 8 4 2 - 3 * - * 2 /
infix-to-postfix conversion:
If the current item is an operand, append it to the result sequence.
else if the current item is (, push it onto the stack
else if the current item is), pop operators off the stack, appending them to the result sequence, until you pop an (, which
you do not append to the sequence
else // the current item is an operator:
If the operator stack is empty, push the current operator onto the stack.
else if the top of the stack is (, push the current operator onto the stack
else if the current operator has precedence strictly greater than that of the operator at the top of the stack,
push the current operator onto the stack
else
pop the top operator from the stack and append it to the result sequence
check again
At the end of the input sequence, pop each operator off the stack and append it to the result sequence
2 8 4 2 - 3 * - * 2 /
operand stack:
2
1:29:16 / 1:44:53

Linked Lists: (doubly linked)

```
struct node
{
  string name;
  node* next;
  node* prev;
}

class myLinkedList
{
public:
  void addtoFront(string name);
  void deleteItem(string name);
  void deleteItem(int slotNum);
  int find(string name);
  void print();
  myLinkedList() //creates empty list
  { first = last = NULL }
  ~myLinked List();
private:
  node* first //beg of list
  node* last  //end of list
```

```
/* You can create linked
lists that are singly linked,
doubly linked, or in a loop
depending on what you need */
```

CHECK THE BOUNDARY CONDITIONS

```
/*inert algrithms that insert
at the top are the easiest to
code and the fastest.
Middle/end are slower/more
complex*/
```

```
/* Destructors must traverse
the entire linked list */
```

Destructing a Derived type
1. Execute the body of the destructor
2. Destroy data members
3. Destroy base part

Linked List Vs. Array
Array is Faster for
- getting a specific item
- less debugging problems
Linked List is Faster for
- inserting at the front
  removing from the middle

Circular Queue: use pointers head and tail to loop around an array

MAKE SURE THE POINTER DOESN'T POINT TO NULL

Constructing a Derived Type
1. Construct base part
2. Construct data members
3. Execut the body of the constructor

```
/* Derived classes can
only access public member
variables and functions of
the base class If you want
Derived classes, but not
the public to access
variables, use protected*/
```

```
/* Copy Constructors and assignment
operators will copy the base and
derived data correctly, UNLESS it is
dynamically allo aited */
```

RECURSION:

1. Identify if the problem is repetitive on a broad scale and/or can be simplified
2. Identify the simplist, complete case
3. Identify the base cases

```
if(base case)
    dosomething
else
    dosomething to reduce the size of
    the problem
```

Inheritance

```
class Base
{
  public
    Base(int p1, int p2)
    void doThis(); //!!!!!
    virtual void doIf(); //default: derived, if it exists
    virtual void doIf2() const =0; //pure virtual
  private:
    [stuff…]
}
class Derived : Public Base
{
  public
    Derived(int p1, int p2) : Base(p1, p2) {}
        //base must be constructed, or default is used
    virtual void doIf2() const;
        //declare overrides virtual as well
    virtual void doIf();
}

void Derived::doIf()
{    Base::doIf2();   }
//to call in a derived class a function from the base
//class that has been overwritten, you need to use
//'Base::'
```

```
/* Recursive functions
should never use
global, static, or
member variables, only
local variables and
parameters! */
```

Generic Programming:
override/define generic comparison operators (<, >, ==, etc)
then, use templates! ☺

Linked Lists: (doubly linked)

```
struct node
{
  string name;
  node* next;
  node* prev;
}

class myLinkedList
{
public:
  void addtoFront(string name);
  void deleteItem(string name);
  void deleteItem(int slotNum);
  int find(string name);
  void print();
  myLinkedList() //creates empty list
  { first = last = NULL }
  ~myLinked List();
private:
  node* first //beg of list
  node* last  //end of list
```

```
/* You can create linked
lists that are singly linked,
doubly linked, or in a loop
depending on what you need */
```

CHECK THE BOUNDARY CONDITIONS

```
/*inert algrithms that insert
at the top are the easiest to
code and the fastest.
Middle/end are slower/more
complex*/

/* Destructors must traverse
the entire linked list */
```

Destructing a Derived type
1. Execute the body of the destructor
2. Destroy data members
3. Destroy base part

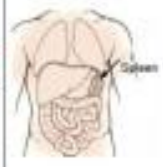
Linked List Vs. Array
Array is Faster for
- getting a specific item
- less debugging problems
Linked List is Faster for
- inserting at the front
  removing from the middle

Circular Queue: use pointers head and tail to loop around an array

MAKE SURE THE POINTER DOESN'T POINT TO NULL

Constructing a Derived Type
1. Construct base part
2. Construct data members
3. Execut the body of the constructor

```
/* Derived classes can
only access public member
variables and functions of
the base class If you want
Derived classes, but not
the public to access
variables, use protected*/
```




```
/* Copy Constructors and assignment
operators will copy the base and
derived data correctly, UNLESS it is
dynamically allo aited */
```

RECURSION:

1. Identify if the problem is repetitive on a broad scale and/or can be simplified
2. Identify the simplist, complete case
3. Identify the base cases

```
if(base case)
    dosomething
else
    dosomething to reduce the size of
    the problem
```

Inheritance

```
class Base
{
  public
    Base(int p1, int p2)
    void doThis(); //!!!!!
    virtual void doIf(); //default: derived, if it exists
    virtual void doIf2() const =0; //pure virtual
  private:
    [stuff…]
}
class Derived : Public Base
{
  public
    Derived(int p1, int p2) : Base(p1, p2) {}
        //base must be constructed, or default is used
    virtual void doIf2() const;
        //declare overrides virtual as well
    virtual void doIf();
}

void Derived::doIf()
{    Base::doIf2();   }
//to call in a derived class a function from the base
//class that has been overwritten, you need to use
//'Base::'
```

```
/* Recursive functions
should never use
global, static, or
member variables, only
local variables and
parameters! */
```

Generic Programming:
override/define generic comparison operators (<, >, ==, etc)
then, use templates! ☺

TEMPLATE CODE:

```
template <typename T>
   //indicates the following class
   //or function is a template
void function(T a[], T p2)
   //T type must be passed as a
   //parameter!
{
    T total = T(); //see*
    ...
}

void function(int a[], int p2)
{...}  //you can write exceptions the
       //compiler will default to

template <typename T1, T2>
    //multi-type templates work too!
void f2(T1 a[], T2 b[])
```

```
/* In templates, the compiler uses
template argument deduction (checks the
parameters) to figure out what functions
to use. Non-template matches have
priority, then to plate matches. If the
call does not match the template exactly,
there will be a compile time error!*/
```

```
/* Using the term T() allows you to
initialize to the "default constructor"
of whatever type you use. For numbers,
this is 0. Bools are false, strings are
empty, chars are the 0 byte. */
```

ALWAYS PLACE TEMPLATES IN THE HEADER FILE

```
/* When you have a function that
traverses the entire leftover list each
time, the algorithm has time complexity
O(N^2): N(N+1)/2 =
1/2N^2+1/2N)*/
```

Template Classes

```
template <typename T>
class something
{…};

template <typename T>
void something<T>::f1(T a)
{…};
```

Inline Functions:

```
/* anything declared inside the class
declaration is automatically inline: the
compiler copies the code wherever you
call the function, speeding up the
program because there's less jumping.
declare external functions inline like
this: */

inline void sclass::f1()
{}

/* setting large functions inline will
greatly increase your exe file size
*/
```

Runtime Time Complexity

```
/*written in terms of "Big
'O' Notation" O(some
function of N), where N is
the number of data terms.
Things to consider if
complexity varies:
Best Case Time
Worst Case Time
Average Case Time
Does your data cause you to
generate the Best/Worst case
often? */
```

```
/* sometimes, for things
like sorting, you consider
complexity of swaps over
comparisons (or some other
specific action) because it
takes significantly longer.
Usually, the longer one is
not swaps, because you
should SWAP POINTERS */
```

INFIX TO POSTFIX

```
Initialize postfix to null
Initialize the operator stack to empty
For each character ch in the infix string
    Switch (ch)
        case operand:
            append ch to end of postfix
            break
         case '(':
            push ch onto the operator stack
            break
        case ')':
              // pop stack until matching '('
            While stack top is not '('
              append the stack top to postfix
              pop the stack
            pop the stack  // remove the '('
            break
         case operator:
            while the stack is not empty and the stack top is not '('
             and precedence(ch) <= precedence(stack top)
                 append the stack top to postfix
                  pop the stack
            push ch onto the stack
            break
While the stack is not empty
    append the stack top to postfix
    pop the stack
```