

ECE 2400 Computer Systems Programming

Topic 13: Object-Oriented Programming

<http://www.csl.cornell.edu/courses/ece2400>
School of Electrical and Computer Engineering
Cornell University

revision: 2026-04-18-22-17

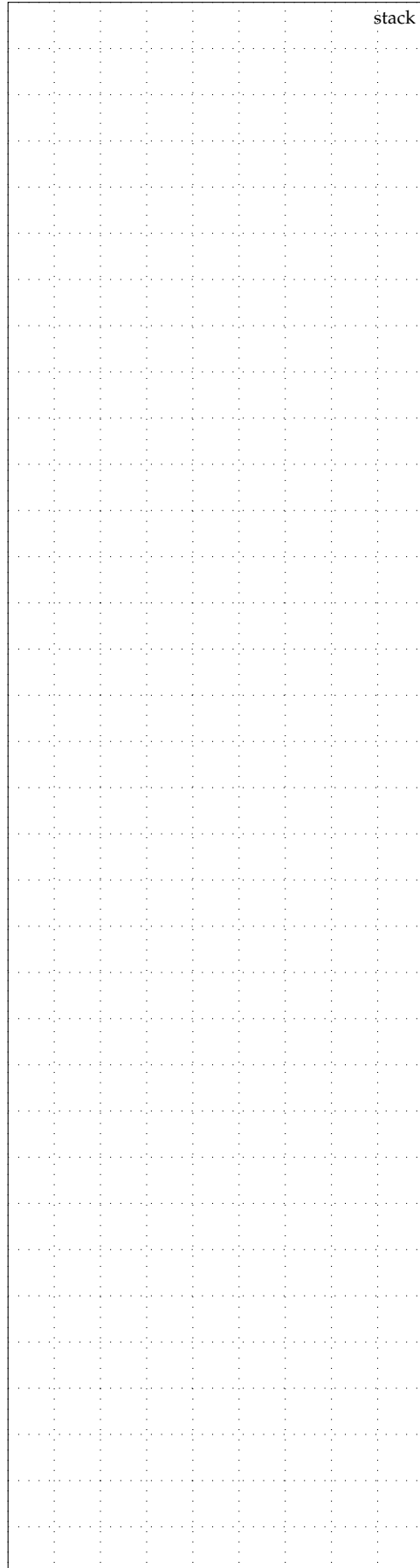
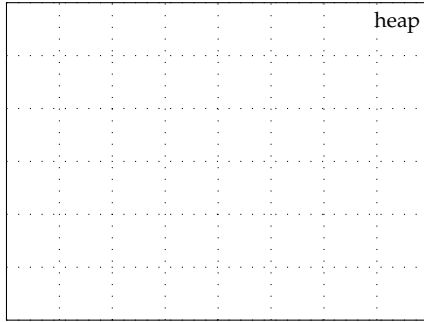
Please do not ask for solutions. Students should compare their solutions to solutions from their fellow students, discuss their solutions with the instructors during lab/office hours, and/or post their solutions on Ed for discussion.

List of Problems

1 Short Answer	2
1.A Constructors and Destructors	3
1.B Accessor Member Function	4
1.C Clock Class	5
1.D Integers in Sign Magnitude Form	6
1.E Filling a Sequence with Zeros	7
1.F Nested Virtual Function Calls	9
1.G Dynamic Polymorphic Array of Objects	10
1.H Shape Interface Inheritance	12
2 Set Data Structures	13
2.A SetLL: Set Implemented with a List of Lists	13
2.B SetLL::contains Complexity Analysis	17
2.C SetAA: Set Implemented with an Array of Arrays	19
2.D SetAA::contains Complexity Analysis	21
2.E Comparing Set Data Structures	22

Problem 1. Short Answer

Carefully plan your solution before starting to write your response. Please be brief and to the point; if at all possible, limit your answers to the space provided.



Part 1.A Constructors and Destructors

The following class Foo includes a single dynamically allocated integer. **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables), explicitly show all constructors and destructors, and fully expand out the stack frames for all function calls.

```

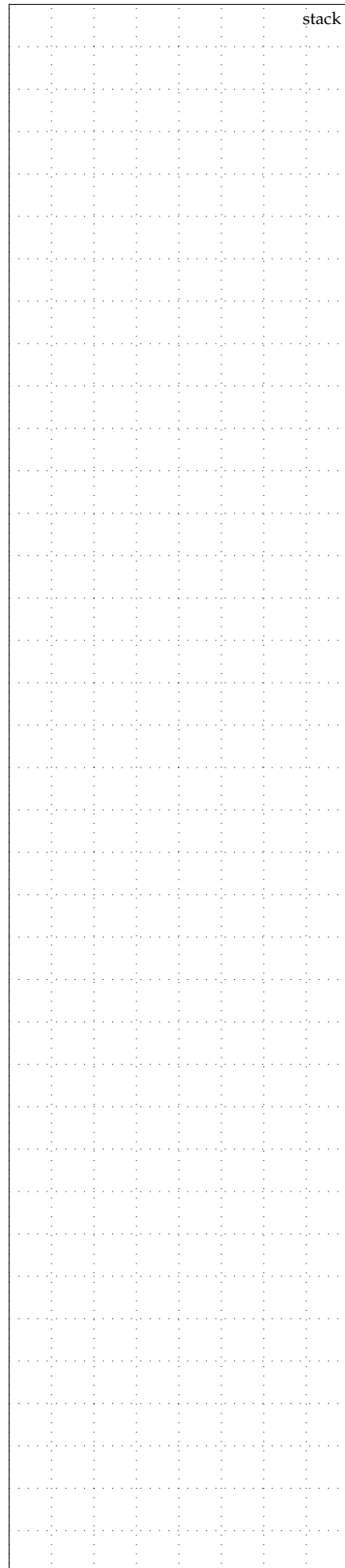
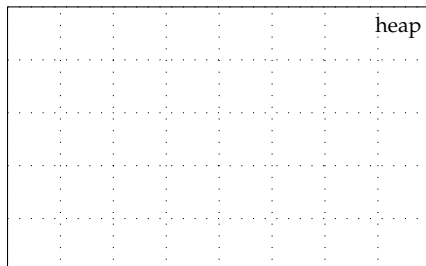
000000 01 class Foo
000000 02 {
000000 03     public:
000000 04
000000 05     Foo() { m_p = nullptr; }
000000 06
000000 07     Foo( int v )
000000 08     {
000000 09         m_p = new int;
000000 10         *m_p = v;
000000 11     }
000000 12
000000 13     Foo( const Foo& x )
000000 14     {
000000 15         if ( x.m_p != nullptr ) {
000000 16             m_p = new int;
000000 17             *m_p = *x.m_p;
000000 18         }
000000 19         else
000000 20             m_p = nullptr;
000000 21     }
000000 22
000000 23     ~Foo() { delete m_p; }
000000 24
000000 25     private:
000000 26         int* m_p;
000000 27 };

000000 01 void bar()      0000 01 int main( void )
000000 02 {                0000 02 {
000000 03     Foo a(3);    0000 03     bar();
000000 04     Foo b(a);    0000 04     return 0;
000000 05 }                0000 05 }
    
```

Part 1.B Accessor Member Function

The following class Foo includes a single dynamically allocated integer, and the class Bar includes a single dynamically allocated instances of Foo. **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables), explicitly show all constructors and destructors, and fully expand out the stack frames for all function calls.

```
01 class Foo
02 {
03     public:
04         Foo() { m_i = new int; }
05         ~Foo() { delete m_i; }
06
07         void set( int v ) {
08             *m_i = v;
09         }
10
11     private:
12         int* m_i;
13 };
14
15 class Bar
16 {
17     public:
18         Bar() { m_f = new Foo; }
19         ~Bar() { delete m_f; }
20
21         void set( int v ) {
22             m_f->set(v);
23         }
24
25     private:
26         Foo* m_f;
27 };
28
29 int main( void )
30 {
31     Bar bar;
32     bar.set( 12 );
33     return 0;
34 };
```



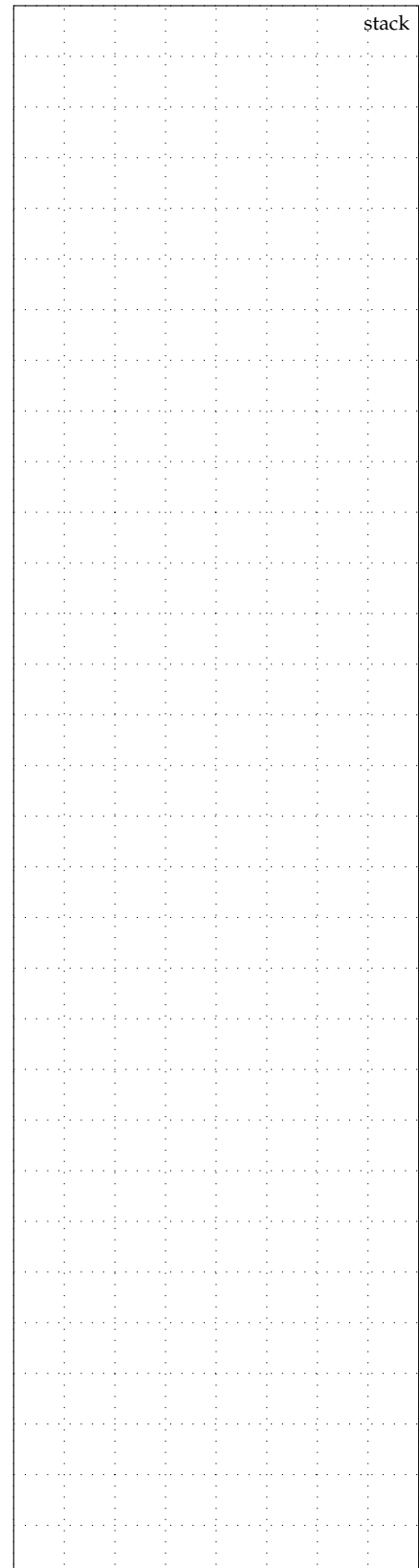
Part 1.C Clock Class

Consider the following C++ program which defines a new class `Clock` to represent a clock with both an hour (`m_hour`) and a minute (`m_min`) field. **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables), explicitly show all constructors and destructors, and fully expand out the stack frames for all function calls.

```

1 style class Clock
2 style {
3 style public:
4 style
5 style   Clock()
6 style   {
7 style     m_hour = 0;
8 style     m_min  = 0;
9 style   }
10 style
11 style   Clock( int hour, int min ) {
12 style     if ( (hour < 0) || (hour > 12)
13 style         || (min < 0) || (min >= 60) )
14 style       throw -1;
15 style
16 style     m_hour = hour;
17 style     m_min  = min;
18 style   }
19 style
20 style   void tick() {
21 style     m_min++;
22 style     if ( m_min == 60 ) {
23 style       m_hour = m_hour + 1;
24 style       m_min  = 0;
25 style     }
26 style     if ( m_hour == 12 )
27 style       m_hour = 0;
28 style   }
29 style
30 style private:
31 style   int m_hour;
32 style   int m_min;
33 style };
34 style
35 style int main( void )
36 style {
37 style   Clock c0;
38 style   Clock c1( 9, 30 );
39 style   c0 = c1;
40 style   c0.tick();
41 style   return 0;
42 style }

```



Part 1.D Integers in Sign Magnitude Form

The following class `Integer` stores integer whole numbers in sign magnitude form (i.e., the `m_pos` field indicates if the number is positive, and the `m_mag` field is the unsigned magnitude). **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables) and fully expand out the stack frames for all function calls, *except you do not need to show any constructors or destructors at all on the stack or execution boxes!* However, you must show the *effect* of these constructors and destructors.

```

000000 01 class Integer
000000 02 {
000000 03 public:
000000 04
000000 05 Integer()
000000 06 {
000000 07     m_pos = true;
000000 08     m_mag = 0
000000 09 }
000000 10
000000 11 Integer( int v )
000000 12 {
000000 13     m_pos = ( v >= 0 );
000000 14     m_mag = m_pos ? v : -v;
000000 15 }
000000 16
000000 17 int get_value() const
000000 18 {
000000 19     int sign = m_pos ? 1 : -1;
000000 20     return sign * m_mag;
000000 21 }
000000 22
000000 23 private:
000000 24     bool        m_pos;
000000 25     unsigned int m_mag;
000000 26 };
000000 27
000000 28 Integer
000000 29 operator+( const Integer& w, int x )
000000 30 {
000000 31     int y = w.get_value();
000000 32     Integer z( x + y );
000000 33     return z;
000000 34 }
000000 35
000000 36 int main( void )
000000 37 {
000000 38     Integer a(-3);
000000 39     Integer b = a + 2;
000000 40     return 0;
000000 41 }

```

stack

Part 1.E Filling a Sequence with Zeros

Recall the C++ singly linked list data structure from lecture. The interface and implementation are shown below with one addition. We have added a new size member function to calculate the number of elements currently stored in the linked list.

```

1 class SListInt
2 {
3 public:
4     SListInt();
5     ~SListInt();
6
7     void push_front( int v );
8     int  size() const;
9
10 private:
11     struct Node
12     {
13         int value;
14         Node* next_p;
15     };
16     Node* m_head_p;
17 };
18
19 SListInt::SListInt()
20 {
21     m_head_p = nullptr;
22 }
23
24 void SListInt::push_front( int v )
25 {
26     Node* new_node_p = new Node;
27     new_node_p->value = v;
28     new_node_p->next_p = m_head_p;
29     m_head_p = new_node_p;
30 }
31
32 SListInt::~SListInt()
33 {
34     while ( m_head_p != nullptr ) {
35         Node* temp_p
36             = m_head_p->next_p;
37         delete m_head_p;
38         m_head_p = temp_p;
39     }
40 }
41
42 int SListInt::size() const
43 {
44     int num_elm = 0;
45     Node* curr_p = m_head_p;
46     while ( curr_p != nullptr ) {
47         num_elm += 1;
48         curr_p = curr_p->next_p;
49     }
50     return num_elm;
51 }
52
53 void fill_with_zeros( SListInt* lst_p,
54                     int N )
55 {
56     while ( lst_p->size() < N )
57         lst_p->push_front(0);
58 }
59
60 int main( void )
61 {
62     SListInt lst;
63     lst.push_front(42);
64     lst.push_front(13);
65     fill_with_zeros( &lst, 10 );
66     return 0;
67 }

```

Consider the following algorithm to fill a SListInt with zeros. Note the list may or may not be initially empty. An example is shown below illustrating the use of this algorithm on a list which initially contains two elements.

Part 1.F Nested Virtual Function Calls

Consider the following three classes: Foo and Bar inherit from IBase. **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables) and fully expand out the stack frames for all function calls, *except you do not need to show any constructors or destructors at all on the stack or execution boxes!* However, you must show the effect of these constructors and destructors.

```

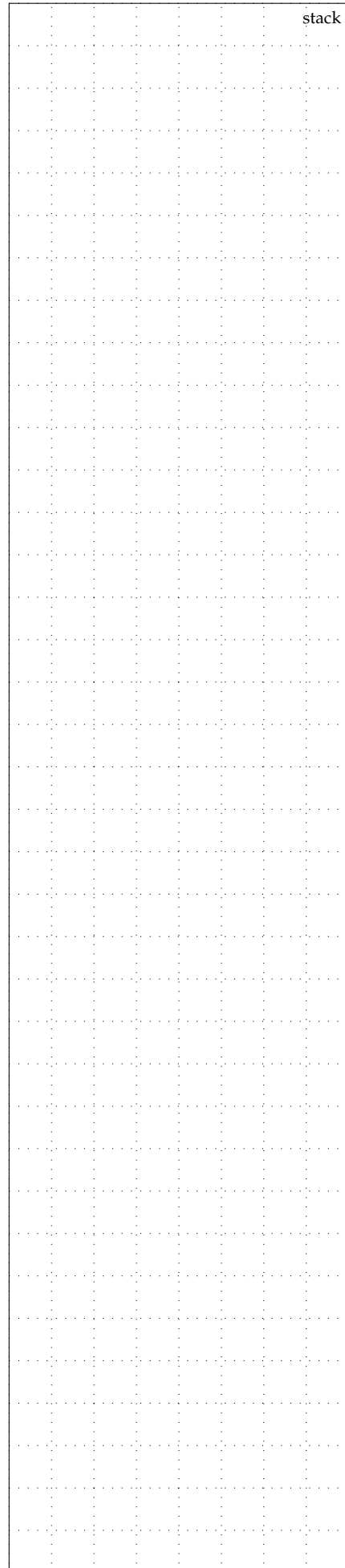
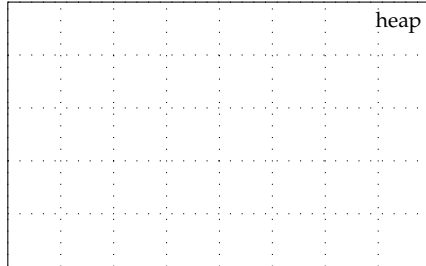
01 class IBase
02 {
03     public:
04         virtual ~IBase() { }
05         virtual void baz( IBase* x ) = 0;
06         virtual void go() = 0;
07 };
08
09 class Foo : public IBase
10 {
11     public:
12         Foo()                { m_d = 0; }
13         void baz( IBase* x ) { x->go(); }
14         void go()           { m_d += 1; }
15
16     private:
17         int m_d;
18 };
19
20 class Bar : public IBase
21 {
22     public:
23         Bar()                { m_d = 10; }
24         void baz( IBase* x ) { x->go(); }
25         void go()           { m_d += 2; }
26
27     private:
28         int m_d;
29 };
30
31 int main( void )
32 {
33     Foo a;
34     Bar b;
35     a.baz(&b);
36     Foo c;
37     c.baz(&c);
38     return 0;
39 }
    
```

stack

Part 1.G Dynamic Polymorphic Array of Objects

The code on the following page illustrates using dynamic polymorphism to create an array of two `I0bjects`. **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables) and fully expand out the stack frames for all function calls, *except you do not need to show any constructors or destructors at all on the stack or execution boxes!* However, you must show the *effect* of these constructors and destructors.

You do not need to show any constructors or destructors at all on the stack or execution boxes!



```
01 class IObject
02 {
03     public:
04         virtual IObject* clone() const = 0;
05         ...
06 };
07
08 class Integer : public IObject
09 {
10     public:
11         Integer( int i ) { m_i = i; }
12         IObject* clone() const {
13             return new Integer( *this );
14         }
15         ...
16     private:
17         int m_i;
18 };
19
20 class Double : public IObject
21 {
22     public:
23         Double( double d ) { m_d = d; }
24         IObject* clone() const {
25             return new Double( *this );
26         }
27         ...
28     private:
29         double m_d;
30 };
31
32 void add( IObject** objs, int idx,
33          const IObject& obj )
34 {
35     objs[idx] = obj.clone();
36 }
37
38 int main( void )
39 {
40     IObject* objs[2];
41     Integer a(5);
42     add( objs, 0, a );
43     Double b(2.5);
44     add( objs, 1, b );
45     return 0;
46 };
```

Part 1.H Shape Interface Inheritance

Consider the following C++ program which uses an IShape class hierarchy similar to lecture. **Draw the state diagram that corresponds to the execution of this C++ program.** You must clearly label all variables in your diagram (including any implicit variables) and fully expand out the stack frames for all function calls, *except you do not need to show any constructors or destructors at all on the stack or execution boxes!* However, you must show the *effect* of these constructors and destructors.

```

000000 01 class IShape
000000 02 {
000000 03     public:
000000 04         virtual void scale( int s ) = 0;
000000 05         ...
000000 06 };
000000 07
000000 08 class Point : public IShape
000000 09 {
000000 10     public:
000000 11         Point()           { m_x = 0;    m_y = 0;    }
000000 12         Point( int x, int y ) { m_x = x;    m_y = y;    }
000000 13         void scale( int s ) { m_x = m_x*s; m_y = m_y*s; }
000000 14         ...
000000 15     private:
000000 16         int m_x; int m_y;
000000 17 };
000000 18
000000 19 class Circle : public IShape
000000 20 {
000000 21     public:
000000 22         Circle()           { m_x=0; m_y=0; m_r=0; }
000000 23         Circle( int x, int y, int r ) { m_x=x; m_y=y; m_r=r; }
000000 24         void scale( int s )         { m_r = m_r*s;    }
000000 25         ...
000000 26     private:
000000 27         int m_x; int m_y; int m_r;
000000 28 };
000000 29
000000 30 void scale_shapes( IShape* s0, IShape* s1, int s )
000000 31 {
000000 32     s0->scale( s );
000000 33     s1->scale( s );
000000 34 }
000000 35
000000 36 int main( void )
000000 37 {
000000 38     Point a(1,2);
000000 39     Circle b(3,4,2);
000000 40     scale_shapes( &a, &b, 2 );
000000 41     return 0;
000000 42 }

```

stack

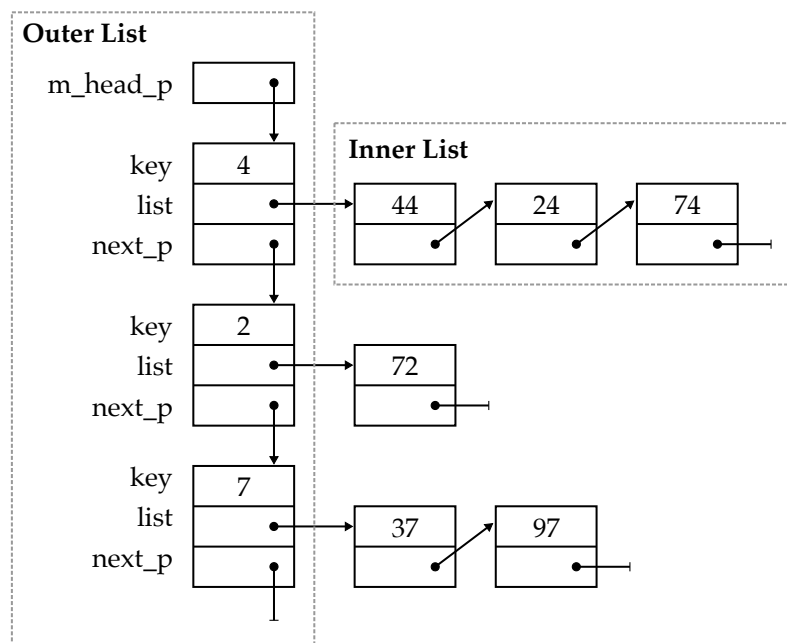
Problem 2. Set Data Structures

In this problem, you will explore two different approaches to implementing a set abstract data type (ADT) for integers. The first approach will use a list of lists, while the second approach will use an array of arrays. Recall that the set ADT should be able to *add* items to the set and to also query if the set *contains* an item.

Part 2.A SetLL: Set Implemented with a List of Lists

The first implementation of the set data structure will use a list of lists. An important aspect of this implementation is that it will rely on the notion of a *key* associated with each integer added to the set. The key k of an integer v is defined as $k = v \% K$ where $\%$ is the C/C++ modulus operator and K is an implementation-defined constant. Recall that the modulus operator essentially returns the remainder of v/K . So for example, if $v = 24$ and $K = 10$, then $k = 4$ (i.e., $24/10$ leaves a remainder of 4) and if $v = 30$ and $K = 8$, then $k = 6$ (i.e., $30/8$ leaves a remainder of 6).

The high-level idea for this implementation is shown below. This example uses $K = 10$, and the set contains $\{ 24, 37, 44, 72, 74, 97 \}$. Each node in the *outer list* contains a *key*, an *inner list* of integers, and a pointer to the next node of the outer list. This implementation should preserve the invariant that all integers with the same key are stored in the same inner list. So we use the following steps when adding integers to the set: (1) check to see if the integer is already in the set; (2) if the integer is not in the set, then we iterate through the outer list to see if the integer's key is already in the set; (3) if the integer's key is in the set, then we simply push the new integer onto the front of the list corresponding to the integer's key; (4) if the integer's key is not in the set, then we add a new node to the outer list and push the new integer onto the front of the list corresponding to this new node.



For the inner list, our implementation will use the standard list data structure for ints which we discussed in lecture. The interface for this data structure is shown below for reference. You should assume the implementation of this list data structure is exactly as described in lecture.

```
1  class ListInt
2  {
3  public:
4
5     ListInt();
6     ~ListInt();
7
8     void push_front( int v );
9
10     class Itr
11     {
12     public:
13         Itr( Node* node_p );
14         void next();
15         int& get();
16         bool eq( const Itr& itr ) const;
17
18     private:
19         Node* m_node_p;
20     };
21
22     Itr begin();
23     Itr end();
24
25 private:
26
27     struct Node
28     {
29         int value;
30         Node* next_p;
31     };
32
33     Node* m_head_p;
34
35 };
36
37 ListInt::Itr operator++( ListInt::Itr& itr, int ); // postfix (itr++)
38 ListInt::Itr& operator++( ListInt::Itr& itr );    // prefix  (++itr)
39 int& operator*( ListInt::Itr& itr );
40 bool operator!=( const ListInt::Itr& itr0, const ListInt::Itr& itr1 );
```

The interface and the private member fields for the set data structure which uses a list of lists is shown below. We have also provided the implementation of the add member function. Study this function closely before continuing with this problem.

```

1  class SetLL
2  {
3  public:
4
5      SetLL();
6      ~SetLL();
7
8      void add( int v );
9      bool contains( int v ) const;
10
11 private:
12
13     // Compile-time constant
14     static const int K = 10;
15
16     struct OuterNode
17     {
18         int         key;
19         ListInt     list;
20         OuterNode* next_p;
21     };
22
23     OuterNode* m_head_p;
24 };

```

```

1  void SetLL::add( int v )
2  {
3      // Key is the remainder of v / K
4      int key = v % K;
5
6      // Step 1. See if set already contains v
7
8      if ( contains(v) )
9          return;
10
11     // Step 2. See if key is in the set
12
13     bool found = false;
14     OuterNode* curr_p = m_head_p;
15     while ( !found && (curr_p != nullptr) ) {
16
17         // Step 3. Key is in the set, add v to list
18         if ( curr_p->key == key ) {
19             curr_p->list.push_front( v );
20             found = true;
21         }
22
23         curr_p = curr_p->next_p;
24     }
25
26     // Step 4. Key is not in set so ...
27     // ... create new outer node
28     // ... add v to the list in new outer node
29
30     if ( !found ) {
31         OuterNode* new_node_p = new OuterNode();
32         new_node_p->key       = key;
33         new_node_p->list      = ListInt();
34         new_node_p->next_p    = m_head_p;
35         m_head_p             = new_node_p;
36
37         // Insert the value into front of new list
38         m_head_p->list.push_front( v );
39     }
40 }

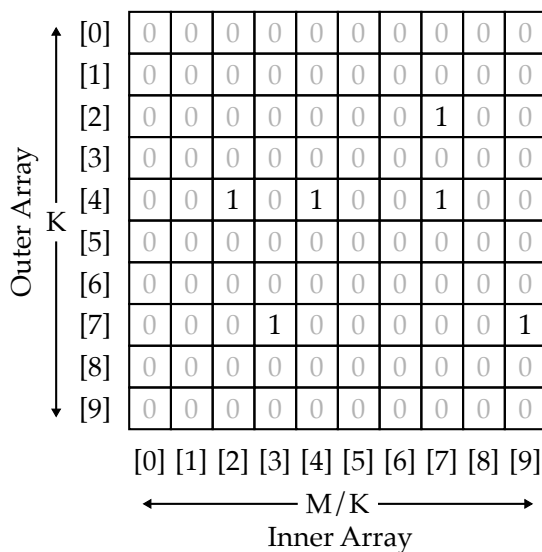
```


Part 2.C SetAA: Set Implemented with an Array of Arrays

The second implementation of the set data structure will use an array of arrays. This implementation will also rely on the notion of a *key* associated with each integer added to the set. The key k of an integer v is defined as $k = v \% K$, exactly as in the previous parts of this problem. This part will also assume that we only wish to add positive integers to the set, and that we know at compile time the maximum integer ($M - 1$) we might *ever* want to add to the set. So there are three key parameters or variables we will need to take into consideration: N is the number of integers currently in the set, K is used to calculate the key for any given integer, and $M - 1$ is the maximum integer we might ever want to add to the set.

The high-level idea for this implementation is shown below. This example uses $K = 10$ and $M = 100$, and the set contains $\{ 24, 37, 44, 72, 74, 97 \}$. There is one element in the data structure for every possible integer we might ever want to add to the set. The value of the element is `true` if the corresponding integer is in the set and `false` if the corresponding integer is not in the set. Each element in the *outer array* is essentially an *inner array*. There are K elements in the outer array and each of the K inner arrays have M/K elements. You can assume that M is evenly divisible by K .

The interface and the private member fields for the set data structure which uses an array of arrays is also shown below.



```

1 class SetAA
2 {
3   public:
4
5     SetAA();
6     ~SetAA();
7
8     void add( int v );
9     bool contains( int v ) const;
10
11   private:
12
13     // Compile-time constants
14     static const int M = 100;
15     static const int K = 10;
16
17     bool m_data[K][M/K];
18
19 };

```


Part 2.E Comparing Set Data Structures

In this problem, you will be qualitatively comparing various search algorithms. **Begin by filling in the following table based on your analysis in this problem.** The time complexity should be the worst-case time complexity for the `contains` member function assume we have N integers in the set, and that the values of these integers are uniformly distributed across the range of 0 to $M - 1$. Recall that for the adaptive implementation, you can ignore the time it takes to periodically create a new list of lists and copy the integers into this new list of lists. The space complexity should capture how much heap space is required to store N unique integers in the set. Your answers should be in terms of N , K , and M as appropriate. For the `SetL` row, assume we implement the set using a single linked list as discussed in lecture.

Implementation	Time Complexity	Space Complexity
SetL (one list)		
SetLL (list of lists, not adaptive)		
SetLL (list of lists, adaptive)		
SetAA (array of arrays)		

Use these results along with deeper insights to perform a comparative analysis of these set data structures, with the ultimate goal of **making a compelling argument for which data structure will perform better across a large number of usage scenarios**. While you are free to use whatever approach you like, we recommend you structure your response in several paragraphs. The **first paragraph** might discuss the performance of `contains` for all four data structure using time complexity analysis. Remember that time complexity analysis is not the entire story; it is just the starting point for understanding execution time. The **second paragraph** might discuss the heap space usage of all four data structures using space complexity analysis. Remember that space complexity analysis is not the entire story; it is just the starting point for understanding space usage. The **third paragraph** might discuss other qualitative metrics such as generality, maintainability, and design complexity. The **final paragraph** can conclude by making a compelling argument for which data structure will perform better in the general case, or if you cannot strongly argue for a single data structure explain why. Your answer will be assessed on how well you argue your position.