MARKING KEY

The University of British Columbia

MARKING KEY

Computer Science 260

Final Examination

8:30 AM, Wednesday, April 18, 2012

Instructor: K. S. Booth **Time: 150 minutes (two hours thirty minutes)** **Total marks: 150**

First _____ Last _____ Student No _____

Printed first name **Printed last name**

Signature _____ Lecture Section **201** Lab Section _____

☛ **This examination has 20 pages + cover. Check that you have a complete paper.**

This is a closed book exam. Notes, books or other materials are not allowed.

Answer all the questions on this paper. The marks for each question are given in { braces }. Use this to manage your time. Good luck.

READ AND OBSERVE THE FOLLOWING RULES:

1. Each candidate should be prepared to produce, upon request, his or her Library/AMS card.

2. No candidate shall be permitted to enter the examination room after the expiration of 10 minutes, or to leave during the first 10 minutes of the examination.

3. Candidates are not permitted to ask questions of the invigilators, except in cases of supposed errors in examination questions.

4. **CAUTION** — Candidates guilty of any of the following, or similar dishonest practices shall be immediately dismissed from the examination and shall be liable to disciplinary action.

   a. Making use of any books, papers or memoranda, calculators or computers, audio or visual cassette players, or other memory aid devices, other than those authorized by the examiners.
   b. Speaking or communicating with other candidates.
   c. Purposely exposing written papers to the view of other candidates. The plea of accident or forgetfulness shall not be received.

Page	Mean	Max
3	8.43	10
4	7.57	10
5	6.98	10
6	6.52	10
7	7.44	10
9	13.03	15
10	7.61	10
11	8.49	10
13	18.16	20
14	5.48	8
16	6.83	12
17	5.23	10
18	8.12	15
Total	109.90	**150**

1. Multiple choice questions { 50 marks — 1 mark per question }

On the next page is a series of short fill-in-the-blanks questions. All of your answers are to be selected from the list below. You may find it convenient to remove this page from the answer booklet so you can look at it while you answer the questions that follow.

1)	abstract	21)	extern	41)	loader	61)	reference
2)	accessor	22)	for	42)	local	62)	Ritchie
3)	activation	23)	function	43)	max-heap	63)	root
4)	aunt	24)	garbage	44)	method	64)	selection
5)	bubblesort	25)	global	45)	min-heap	65)	shallow
6)	Budd	26)	head	46)	module	66)	signature
7)	child	27)	heap	47)	mutator	67)	stack
8)	circular	28)	heapsort	48)	next	68)	static
9)	code	29)	helper	49)	node	69)	Stroustrup
10)	compiler	30)	heterogeneous	50)	null	70)	subclass
11)	constant	31)	homogeneous	51)	overload	71)	superclass
12)	debugger	32)	Horstmann	52)	override	72)	tail
13)	declaration	33)	initialization	53)	parent	73)	template
14)	deep	34)	initializer	54)	previous	74)	trichotomy
15)	depth	35)	iterator	55)	private	75)	v-pointer
16)	destructor	36)	Kernighan	56)	protected	76)	v-table
17)	diddly	37)	leak	57)	public	77)	value
18)	do wop	38)	level	58)	pure	78)	vector
19)	dummy	39)	linker	59)	quicksort	79)	virtual
20)	dynamic	40)	literal	60)	recursion	80)	width

Each statement will have one ■■■■■■■■ within it, which is where the missing term or phrase would appear. Choose the best answer from among those above and write its number in the space provided in the first column. Do not write the term or phrase. It may be a good idea to read over the list of terms and phrases before you start answering. Some of the terms listed may not appear in any of the statements, some may appear in more than one statement, and some many appear in exactly one statement.

Continue on to the next page…

You may remove this page from the exam booklet.

Read the instructions on the previous page. Enter the number for your answer in the first column. Do not write words!

(a)	**70**	A derived class is a [redacted] class of a base. subclass
(b)	**71**	A base class is a [redacted] class of a derived class. superclass
(c)	**51**	A method in a derived class that has the same name but has a different signature than a method in a base class will [redacted] the method in the base class. overload
(d)	**52**	A method in a derived class that has the same name and the same signature as a method in a base class will [redacted] the method in the base class. override
(e)	**58**	An abstract method is a [redacted] virtual method. pure
(f)	**58**	A class is abstract if it has one or more [redacted] virtual methods. pure
(g)	**79**	The method declared here is [redacted]. `virtual void foo();` virtual
(h)	**75**	In a class that has one or more virtual methods every object has a [redacted] stored within the object. v-pointer
(i)	**76**	A class has one or more virtual methods will have exactly one [redacted] that is stored separately from all of the objects in the class. v-table
(j)	**46**	A class in the C++ programming language corresponds to the [redacted] concept in some programming languages. module

This question continues on the next page…

Put a slash through every wrong answer, and an X through every blank answer. Total the number of correct answers and write at top of page AND on first page.

(This question is continued from the previous page.

(k)	**30**	Structs are usually ■■■■■■■■ data types. heterogeneous
(l)	**31**	Arrays are always ■■■■■■■■ data types. homogenous
(m)	**68**	The ■■■■■■■■ keyword is used to indicate that a function declared inside of a class declaration will not have an implicit **`this`** parameter in its stack frames. static
(n)	**21**	The ■■■■■■■■ keyword is used to indicate that the scope of a variable declared outside of any class declaration or block may have its definition specified in some other compilation unit than the one in which this declaration appears. extern
(o)	**68**	The ■■■■■■■■ keyword is used to indicate that the scope of a variable declared inside a block has local scope but does not have automatic extent. static
(p)	**68**	The ■■■■■■■■ keyword is used to indicate that a variable declared inside a class declaration is not an instance variable within objects of the class. static
(q)	**44**	A member function in a class is often referred to by the term "■■■■■■■■" to distinguish it from a non-class function. method
(r)	**34**	The ■■■■■■■■ list in the following constructor definition ensures that the member variable **`count`** has a value before the object's constructor is invoked. `InSet::InSet() : count(0) {}` initializer
(s)	**29**	■■■■■■■■ functions are member functions that are usually only called by other member functions to perform actions that are required in more than one member function that clients invoke. helper
(t)	**23**	The type of the parameter in the following function declaration is ■■■■■■■■. `char foo( int (*bar) (double))` function

This question continues on the next page…

Put a slash through every wrong answer, and an X through every blank answer. Total the number of correct answers and write at top of page AND on first page.

This question is continued from the previous page.

(u)	**25**	■■■■■■■■ variables have static extent and are visible to all compilation units . global
(v)	**42**	■■■■■■■■ variables have automatic extent and are not visible to other compilation units. local
(w)	**20**	■■■■■■■■ variables are stored in the heap segment. dynamic
(x)	**43**	The ■■■■■■■■ data structure can be used to sort the elements in an array into increasing order without using more than O(1) additional storage. max-heap
(y)	**45**	The ■■■■■■■■ data structure can be used to keep track of the next element to be served in a priority queue in which large numbers are served later than small numbers. min-heap
(z)	**73**	The **`vector`** class has a ■■■■■■■■ declaration that allows it to be used as a container for any type of data desired within a program. template
(aa)	**61**	Call-by-■■■■■■■■ avoids the need to make a copy of a parameter. reference
(bb)	**60**	Two functions that invoke each other is one example of ■■■■■■■■. recursion
(cc)	**13**	Two functions that invoke each other will require that at least one of them have a forward ■■■■■■■■ so the compiler knows how to invoke it from the other. declaration
(dd)	**33**	■■■■■■■■ of a variable can use syntax similar to assignment but it is definitely not the same as the assignment operator if the variable is an object in a class. initialization

This question continues on the next page…

Put a slash through every wrong answer, and an X through every blank answer. Total the number of correct answers and write at top of page AND on first page.

(This question is continued from the previous page.)

(ee)	**59**	The ■■■■■■■■■ algorithm partitions the elements of an array using left and right pointers as part of its work sorting the elements. quicksort
(ff)	**28**	The ■■■■■■■■■ algorithm is guaranteed to sort all of the *n* elements in an array using at most O(*n* log *n*) comparisons. heapsort
(gg)	**59**	The ■■■■■■■■■ algorithm sorts all of the *n* elements in an array using at most O(*n* log *n*) comparisons on average, but its worst case complexity is O(n^2). quicksort
(hh)	**69**	Bjarne ■■■■■■■■■ designed the C++ programming language to be an object-oriented extension to the C programming language. Stroustrup
(ii)	**6**	Timothy ■■■■■■■■■ is one of the co-authors of *Big C++, 2nd Edition*. Budd
(jj)	**32**	Cay ■■■■■■■■■ is one of the co-authors of *Big C++, 2nd Edition*. Horstmann
(kk)	**66**	Whether a function is a method or a regular function, and the class it belongs to if it is a method, are all part of the ■■■■■■■■■ for the function. signature
(ll)	**2**	The **`const`** keyword appears at the end of the declaration for all ■■■■■■■■■ methods in a class. accessor
(mm)	**47**	No ■■■■■■■■■ method can be invoked on an object that is declared to be **`const`**. mutator
(nn)	**16**	As for a constructor, the ■■■■■■■■■ has no return type (not even **`void`**) permitted in its declaration. destructor

This question continues on the next page…

Put a slash through every wrong answer, and an X through every blank answer. Total the number of correct answers and write at top of page AND on first page.

This question is continued from the previous page.

(oo)	**54**	In a linked list data structure the pointer from the element that is immediately after another is often called the ■■■■■■■■■ pointer. previous
(pp)	**48**	In a linked list data structure the pointer from the element that is immediately before another is often called the ■■■■■■■■■ pointer. next
(qq)	**50**	In a linked list data structure a pointer whose value is zero is the ■■■■■■■■■ pointer that might be in the last element at the end of the list. null
(rr)	**26**	The first elelment in a linked list data structure is often called the ■■■■■■■■■ of the list. head
(ss)	**19**	Insertion and deletion from a listed data structure are often easier if one or more ■■■■■■■■■ elements that have no information stored in them are at the beginning and end. dummy
(tt)	**8**	A linked list data structure in which the first element points "backward" to the last element and the last element points "forward" to the first element is ■■■■■■■■■. circular
(uu)	**49**	In a tree data structure the information is usually stored in the ■■■■■■■■■ elements that are linked together with pointers. node
(vv)	**63**	The ■■■■■■■■■ of a tree data structure is the top-most element. root
(ww)	**7**	An element in a tree data structure that is pointed downward to by an element above it is the ■■■■■■■■■ of the upper element. child
(xx)	**53**	The ■■■■■■■■■ of an element in a tree data structure points downward to it and possibly to other elements as well. parent

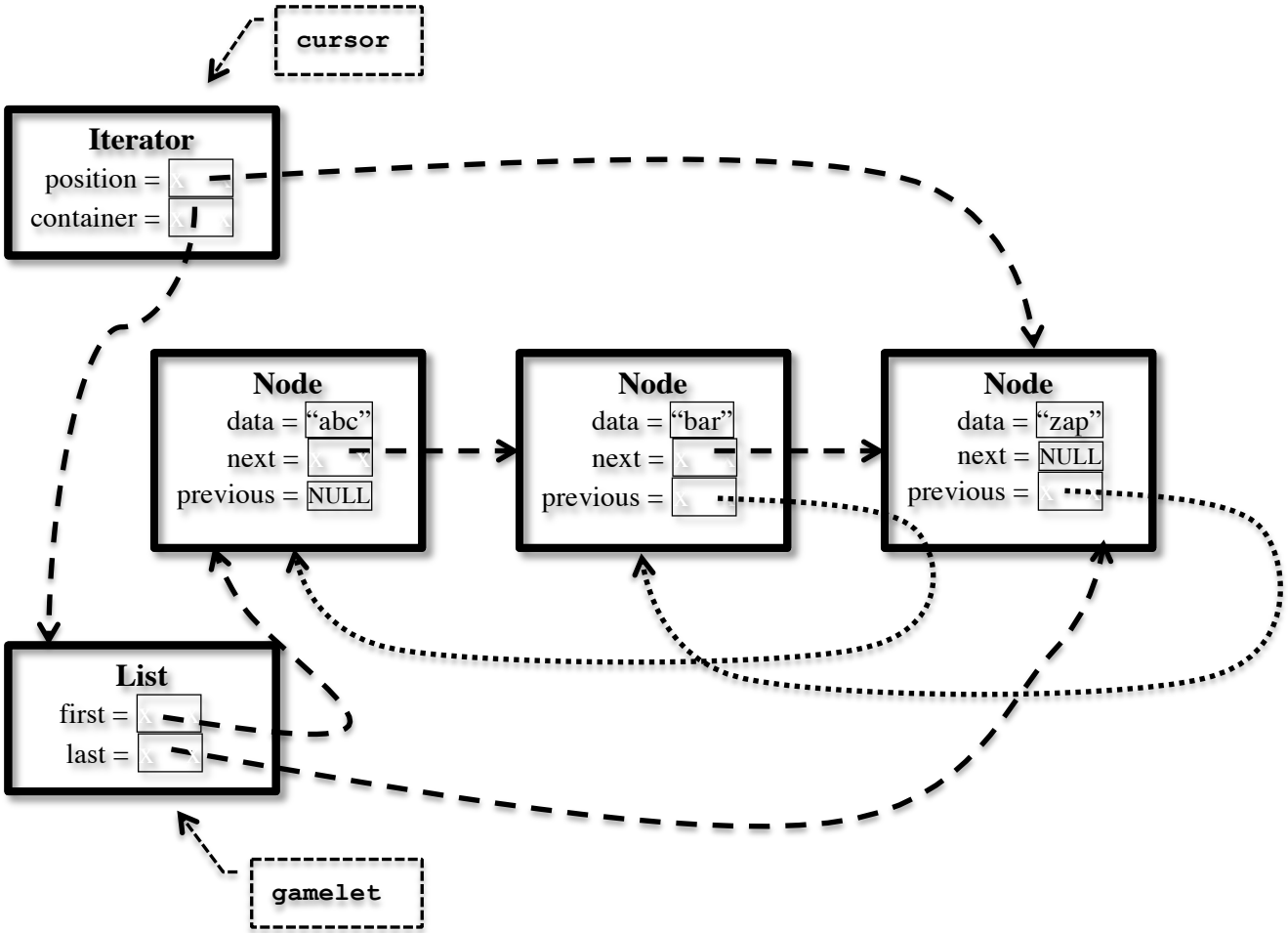
Put a slash through every wrong answer, and an X through every blank answer. Total the number of correct answers and write at top of page AND on first page.

2. Linked lists { 15 marks }

The source code to manipulate a doubly-linked list using iterators is described in the textbook, *Big C++*, and was part of Assignment 3. The diagram below shows the values in the **`gamelet`** list and **`cursor`** iterator objects and all of the values in dynamic storage that they point to after the following code is executed.

```
List gamelet;
gamelet.push_back( "abc" );
gamelet.push_back( "zap" );
Iterator cursor = gamelet.begin();
cursor.next();
gamelet.insert( cursor, "bar" );
```

Use the diagram as a guide in answering the question on the next page.



Continue on to the next page…

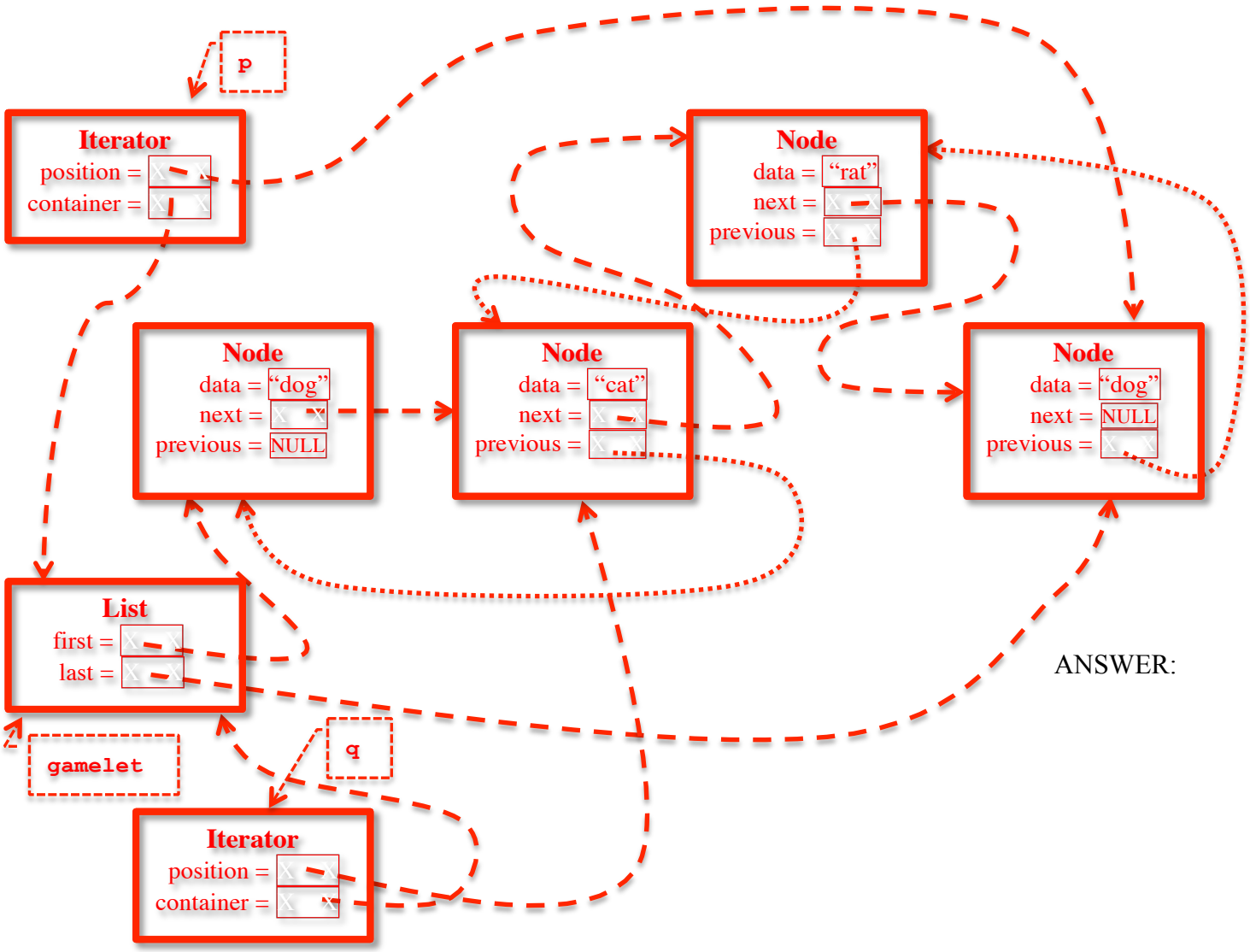
You may remove this page from the exam booklet.

The code below also uses **`List`** and **`Iterator`**. The **`begin()`** method returns the position of the first element in a list; **`end()`** returns a position after the last element in the list (so it is "null").

```
List gamelet;
gamelet.push_back( "cat" );
gamelet.push_back( "dog" );
Iterator p = gamelet.begin();
p.next();
gamelet.insert( p, "rat" );
Iterator q = gamelet.end();
q.previous();
q.previous();
q.previous();
gamelet.insert( q, p.get() );
```

Put a slash through every wrong answer, and an X through every blank answer. Total the number of correct answers and write at top of page AND on first page.

Draw a diagram, similar to the diagram on the previous page, that shows the values in the **`gamelet`** list and the two iterator objects **`p`** and **`q`** and all of the values in dynamic storage that they point to after the code above has been executed.



ANSWER:

3. Class declarations and method definitions with dynamic memory { 20 marks }

Refer to the following class declaration for all parts of this question. It is part of a complete class declaration for **`IntSet`** that is similar to what we have seen in lectures and on the first assignment. In your answers use only methods and variables that are shown here (not others that may exist).

```
class IntSet {
  private:
    static const int CAPACITY = 4;
    static const int RATIO = 3;
    void grow();          // Expands capacity by a factor of RATIO
    int capacity_;
    int *data_;
    int count_;
      . . .
  public:
      . . .
    bool find( int entry ) const;
    int get( int index ) const;
    int size() const;     // returns the number items in the IntSet
};
```

(a) { 10 marks } Write a complete definition for the helper method **`grow()`** based on the information in the declaration above and the example in the first assignment. When **`grow()`** has finished its work, the object should be able to hold more items than it did previously. Be sure to avoid a memory leak.

ANSWER:

```
void IntSet::grow()
{
   int* temp = new int[RATIO*capacity_];
   for (int i=0; i<count_; i++ )
   {
     temp[i] = data_[i];
   }
   delete [] data_;
   data_ = temp;
   capacity_ *= RATIO;
}
```

The code you were asked to write came verbatim from code that was given to you on Assignment 1.

½ mark off for each missing or wrong item

- return value type must be void
- IntSet:: is required
- grow() with no parameters
- must get a new array of int
- size of new array should be RATIO * size of old array
- for loop to copy values from data_ to temp
- look limits should be zero to capacity_ or count_
- must delete old data_
- delete must have []
- must set data_ to point to new array
- must set new value to capacity_
- sometimes order makes a different (check for this)

Continue on to the next page…

(b) { 10 marks } Two sets are equal if and only if they contain exactly the same elements. Write a C++ function definition that determines whether two **`IntSet`** objects are equal. It should return **`true`** if they are, otherwise it should return **`false`**. The first line of the function definition is provided for you. Note that the function is not a friend of the class, so you cannot access any private members of the class.

ANSWER:

```
bool operator==( const IntSet& A, const IntSet& B )
{
  if ( A.size() != B.size() ) return false;
  for ( int i=0; i<A.size(); i++ )
  {
    if ( !B.find( A.get( i ) ) ) return false;
  }
  return true;
}
```

RANT

The function returns a bool. Don't use 0 and 1 to represent false and true. Use the built-in constants "true" and "false". This is a course in C++, not in C.

-2 if the code tries to look at member variables. This is function, not a method, and it is not a friend of the class. So it has to use accessors because it cannot look at member variables. The objects are const, to mutators cannot be used.

-1/2 for each return case that is wrong (three of them are usually needed.)

-1/2 if size of sets not checked

-2 if the code tries to look at member variables. This is function, not a method, and it is not a friend of the class. So it has to use accessors because it cannot look at member variables. The objects are const, to mutators cannot be used.

-1/2 for each return case that is wrong (three of them are usually needed.)

-1/2 if size of sets not checked properly

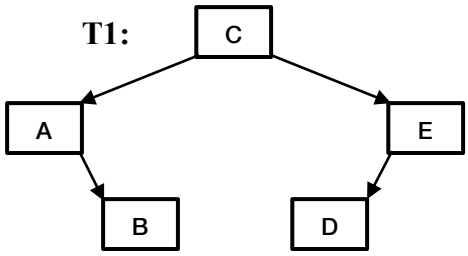
-1/2 if loop iteration count is not right

-1 if loop does not do a sensible job of checking to see if A is strictly contained in B. What many students missed was getting the case A==B correct

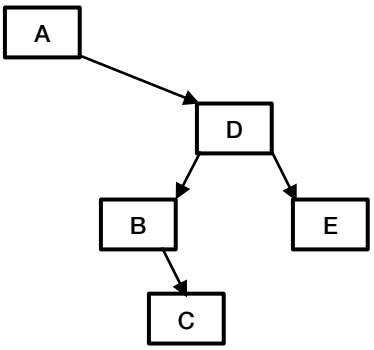
4. Binary Search Trees { 20 marks }

The five letters **`A-B-C-D-E`** are inserted into an empty binary search tree (BST) in some order. The trees shown below are examples. Refer to these when you answer all parts of this question on the next page.

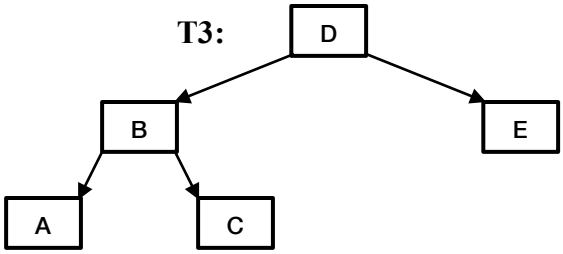
T1:



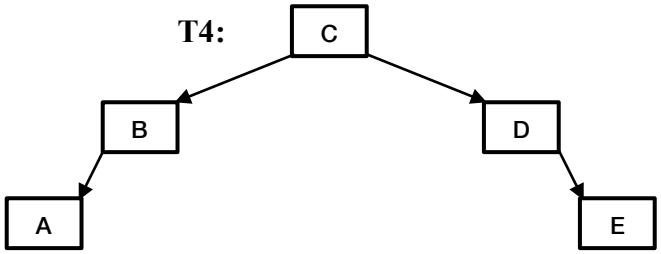
T2:



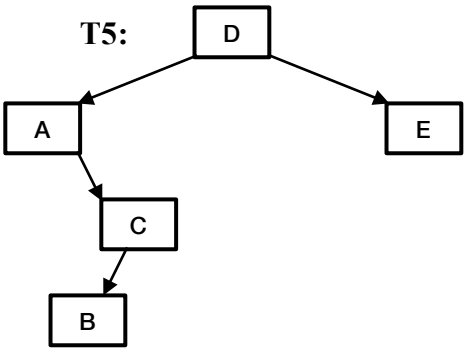
T3:



T4:



T5:



Continue on to the next page…

You may remove this page from the exam booklet.

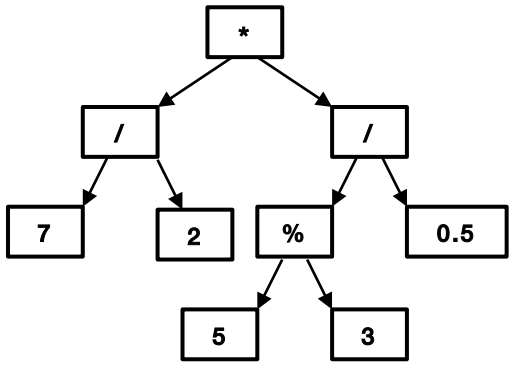
For each statement below, determine the subset of the trees shown on the previous page for which the statement is true and indicate those by writing the letter '**T**' in the corresponding column. The first part has been completed for you to illustrate the format for your answers. Do not write anything in a column if the statement is false for that tree.

Statement	T1	T2	T3	T4	T5
(a) The root of this tree contains the letter '**C**'.	**T**			**T**	
(b) There is a leaf node in this tree that contains the letter '**C**'.		**T**	**T**		
(c) The height of this tree is 3.		**T**			**T**
(d) There is only one node at level 1 in this tree.		**T**			
(e) There are exactly two nodes at level 1 in this tree.	**T**		**T**	**T**	**T**
(f) There are exactly two nodes at depth 2 in this tree.	**T**	**T**	**T**	**T**	
(g) There is exactly one node at height 2 in this tree.	**T**	**T**	**T**	**T**	**T**
(h) There are multiple nodes each with two children in this tree.			**T**		
(i) The preorder traversal of this tree is "**DBACE**".			**T**		
(j) The postorder traversal of this tree is "**BADEC**".	**T**				
(k) The inorder traversal of this tree is "**ABCDE**".	**T**	**T**	**T**	**T**	**T**

0.4 marks per answer (10 rows of 5 cells each).
Put a slash through every wrong answer, and a "T" in every blank answer that should be true. Total the number of correct answers and write at top of page AND on first page.

5. Binary expression trees { 8 marks }

Consider the following binary expression tree when answering each part of this question. Assume that the expression is evaluated using the standard C and C++ rules for arithmetic expressions and type conversions.



(a) { 2 marks } What is the preorder expression for the tree?

```
* / 7 2 / % 5 3 0.5
```

(b) { 2 marks } What is the postorder expression for the tree?

```
7 2 / 5 3 % 0.5 / *
```

Only exactly correct answers receive marks.

Solutions with parentheses receive zero marks.

Preorder and postorder (both of which are Polish notation) are used precisely because they require no parentheses!

(c) { 2 marks } Assuming the precedence rules for C and C++, what is the inorder expression that has the least number of parentheses that correctly represents the expression corresponding to the tree?

```
7 / 2 * (5 % 3 / 0.5)
```

Additional parentheses inside the denominator are not required because % and / have the same precedence and thus will be done left-to-right. One mark is lost if there are any extra parentheses.

(d) { 2 marks } What is the value of the expression represented by the tree?

```
12.0
```

2.0 marks: 12.0

1.5 mark 12

1.0 mark: 14.0

0.5 mark: 14

The first division is integer division because the expression has not yet been converted to floating point. Similarly, the remainder operation is also done with integers, but the second division is floating point because its second operand is explicitly floating point. The multiplication is thus also floating point because one of its operands is a floating-point value. No partial credit is given for incorrect answers or for unevaluated expressions.

1/2 mark is lost if there is no decimal point (the zero is optional).

Refer to the following C++ source code for all of the remaining questions on the exam. Use the line numbers to label the location of occurrences of identifiers in the source code.

```
 1  #include <iostream>
 2
 3  using namespace std;
 4
 5  double x = 5.0;
 6
 7  class Ralph {
 8    friend ostream& operator<<( ostream&, const Ralph& );
 9    private: double *x; // this will point to array of doubles
10    public:  Ralph( double z ) : x(new double[int(z)]) { }
11             ~Ralph() { delete x; } // the destructor should be virtual
12             static double foo( double z ); // no "this" and not virtual
13  };
14
15  class George : public Ralph {
16    private: double *x; // this will point to a single double
17    public:  George( double z ) : Ralph( z ), x(new double(z*z)) {}
18             ~George() { delete x; }  // This should be: delete [] x;
19             double foo( double z );  // This should be const
20  };
21
22  double Ralph::foo( double z ) { return ::x += z; }
23
24  double George::foo( double z ) { return *x += z; }  // const here too!
25
26  double foo( double &z ) { z = x * x;  return x; }
27
28  ostream& operator<<( ostream& out, const Ralph& object )
29    { out << *object.x;}
30
31  int main() {
32    double &z = x;
33    static double x = 3.0;
34    Ralph w(x); // allocates 3 doubles in Ralph()
35    Ralph *p = new George (z); // 2 ptrs, 1 double, and 5 doubles
36    p->foo( w.foo( foo(x) ) ); // dynamic binding used for p->foo()
37    cout << *p << endl;
38    delete p;
39  }
```

The foo() method in Ralph is static, so it does not involve an object. The foo() method in George does not change any member variables, so it could (and should) be "const" (see above).
The destructor ~Ralph() should be virtual so line 38 calls the correct destructor.
The destructor ~George() should delete an array (this does not affect the answers).

Continue on to the next page…

You may remove this page from the exam booklet.

6. Scope and extent rules in C++ { 12 marks }

In C++ each occurrence of an identifier in the program on page 15 is bound to a variable or function that has been declared. The binding is determined by the scope rules. Fill in the following table with the line number of the declaration for the variable or function to which an identifier is bound, and the scope and extent, for each of the indicated occurrences of variables in the program. The line number and identifier are in the first two columns. If the extent cannot be determined write **"*it depends*"**. For functions, write the name of the class in which they are members instead of the extent if they are methods; otherwise for non-member functions write **"*function*"** instead of the extent.

Line	Identifier	Declared	Scope	Extent
10	`x`	9	class	*it depends*
17	`x`	16	class	*it depends*
22	`x`	5	global	static
24	`x`	16	class	*it depends*
26	`x`	5	global	static
29	`x`	9	class	*it depends*
35	`z`	32	local	automatic
36	`w`	34	local	automatic
36	`x`	33	local	static
36	`foo` (left occurrence)	12	class	Ralph
36	`foo` (middle occurrence)	12	class	Ralph
36	`foo` (right occurrence)	26	global	*function*

1/3 off for each wrong entry.

7. Stack frames and object memory requirements { 10 marks }

The five diagrams below represent the words in a stack frame or an object that might exist during the execution of the program on page 15. Each entry is either a four-byte word or an eight-byte double word, as indicated within the boxes.

B1

double
address of a double
address in code segment

B2

address of a double
address of a double
address in code segment

B3

address of a double

B4

double
address of an object
address of a double
address in code segment

B5

address of a double
address of a double

In the second column of the table below write one of the two-letter names for the diagram above that best describes the memory block for the five stack frame or object types listed in the first column of the table.

Stack frame or object type	Memory diagram
Stack frame for the function defined on line 22	**B1**
Stack frame for the function defined on line 24	**B4**
Stack frame for the function defined on line 26	**B2**
Object in the `Ralph` class	**B3**
Object in the `George` class	**B5**

8. Dynamic storage allocation { 15 marks }

Each time the **`new`** operator is invoked during the execution of the program on page 15 the storage allocator must provide a block of memory and return a pointer to the block. The number of bytes that are allocated will depend on the type of the variables that require memory and whether they are arrays.

Each part of this question requires only a short answer (a number or YES/NO). Include an explanation for your answer if you want to get partial marks for incorrect answers.

(a) { 3 marks } How many distinct times is the storage allocator asked to provide a block of memory from the heap during the program on page 15's entire execution?

ANSWER: 4 times (When w is created the Ralph constructor allocates an array of 3 doubles, and when the George object is created the new operator is invoked once for the object (2 pointers) and then once each by both the Ralph and the George constructors for 5 doubles and 1 double.)

4 = 3 marks, 3 = 2 marks, 2 = 1 mark, if the rationale is correct.

(b) { 3 marks } How many total bytes are allocated on the heap during the program's entire execution?

ANSWER: 80 bytes (The Ralph constructor for w allocates 24 bytes for 3 doubles; 8 bytes for two pointers are allocated when the object pointed to by p is created. The George constructor allocates 1 double after the Ralph constructor allocates 5 doubles, so there are 56 bytes for the second object and 24 for the first.)

(c) { 3 marks } Does the program on page 15 have any memory leaks? (After all of the destructors have been invoked for variables declared in **`main()`**, is there any allocated memory that has not been returned to the storage allocator?)

ANSWER: YES (The destructor for Ralph is not virtual so static binding is used, which means that the destructor for George is not invoked and thus the memory allocated on the heap by the George constructor is never returned to the storage allocator.)

(d) { 3 marks } Does the program on page 15 have any dangling pointers? (The pointer **`p`** is not considered a dangling pointer: it disappears right after it is deleted because the block it is in exits.)

ANSWER: NO (The failure to use [] on line 18 does not leave a dangling pointer or a memory leak. It might cause a run-time error if the allocator checks for this, but if there is no check all of the memory will be freed. Because the array has no objects (just doubles) there is not a problem with not calling destructors.

(e) { 3 marks } Does the program on page 15 make any illegal memory accesses?

ANSWER: NO

Continue answers here – identify answers by writing the starting page number!

Instructions for marking

Continue answers here – identify answers by writing the starting page number!

Instructions for marking