

CS 216 Exam 1 – Spring 2005 - **SOLUTION**

Name: _____ Lab Section: _____

Email Address: _____

This exam is **closed note, closed book**. You will have an hour and fifty minutes total to complete the exam. You may **NOT** use calculators.

It is an Honor Code violation to discuss this exam with ANYONE (**including** other students who have **already taken** the exam) until after 9:30pm Tuesday, Feb 22, 2005

Good Luck!!

	MAX	SCORE
TOTAL	97	

Write and sign **pledge** after taking the exam:

CS 216 Exam 1

1) **Define** each term and give an **example** that explains it:

a) (2 points) hidden bit in IEEE floating point

Represents the highest order bit in the mantissa. This is a bit that is not actually stored in IEEE floating point representation. For example, it is assumed that the mantissa always starts with 1.x where only the x part is actually stored. (A picture would be good here.)

b) (2 points) deque (NOT dequeue)

A deque is a double-ended queue. Values can be added and removed from both the front and the back of the queue. (A picture would be good here.)

2) (3 points) List the 3 functions that make up the gang of three

copy constructor, operator=, destructor

3) (3 points) Describe WHY you would need to implement these 3 functions yourself. Write a class declaration for a class that you should implement the gang of three for.

A class that has a pointer as one of its data members is usually an example of a class that would require you to implement your own version of the gang of 3. In particular, a class where you call new to allocate memory off of the heap somewhere in the code for the class – maybe in the constructor, maybe in an insert function, as we did in the linked list class from lab 1 – is an example of a class where you most likely should implement your own gang of three.

If you do not implement your own versions of the copy constructor and operator= then what the compiler gives you by default is a shallow copy. This may be o.k. in some situations but in the linked list example from lab 1, shown partially below, it means that if A and B are both of type List, then a shallow copy used for A=B will mean in effect that A now refers to the same list that B does, so that modifications to B will also change A (this is probably not what the programmer expects!) In addition, if you do not implement your own destructor, then the one provided by the compiler will only delete the first node in the linked list – leaving the remainder of the list still allocated – a memory leak.

```
class List {
public:
    . . .
private:
    Node *head;
}
```

4) (3 points) Name 3 factors that are ignored by big-Oh notation.

Hardware, Operating system, compiler, programming language used, assumes that all operations take the same amount of time

Other answers that were accepted: ignores constant factors, throws out lower ordered terms, (these were described as big-Oh math), also does not work for small values of N (although this is not really a factor that it ignores, big-Oh just doesn't give you a good estimate of run-time/memory use for small values of N)

5) (4 points) Fill in the blanks in the definition of big-Oh notation:

$T(N) = O(f(N))$ if:

there are positive constants c and n_0 such that:

$$T(N) \leq c * f(N)$$

when:

$$N \geq n_0$$

- 6) (10 points total) Describe the running time of the following pseudocode in Big-Oh notation in terms of the variable n . Assume all variables used have been declared. **Show your work for partial credit.**

```
int foo(int k) {
    int cost;
    for (int i = 0; i < k; ++i)
        cost = cost + (i * k);
    return cost;
}
```

a) `answ = foo(n);` $O(n)$

b) `int sum;`
`for (int i = 0; i < n; ++i)` $O(n^2)$
 `if (n < 1000)`
 `cout << "cool!";`
 `else`
 `sum += foo(n);`

c) `for (int i = 0; i < n * 1000; ++i) {` $O(n^2)$
 `sum = (sum * sum) / (n * i);`
 `for (int j = 0; j < i; ++j)`
 `cout << j * i;`
`}`

d) `for (int i = 0; i < n + 100; ++i) {` $O(n^3)$
 `for (int j = 0; j < i * n; ++j)`
 `sum = sum + j;`
 `for (int k = 0; k < n + n + n; ++k)`
 `c[k] = c[k] + sum;`
`}`

e) `for (int j = 4; j < n; j=j+2) {` $O(n^3)$
 `cin >> val;`
 `for (int i = 0; i < j; ++i) {`
 `cout << val + i * j;`
 `for (int k = 0; k < n; ++k)`
 `val++;`
 `}`
`}`

7) (6 points total) What is the representation of each of the following in the indicated radix? Be sure to show your work.

a) 23_9 in decimal **21_{10}**

b) 126_8 in hex **56_{16}**

c) $1F_{18}$ in radix 10 **33_{10}**

8) (6 points total) Consider the positive binary integer represented in two's complement:

0010110110100101_2 .

a) Express this binary number in octal

26645_8

b. Express this binary number in hexadecimal

$2DA5_{16}$

c. Negate the number (i.e. give the two's complement representation of a negative version of the same number) Use the same number of bits.

$1101\ 0010\ 0101\ 1011$

9) For each operation below give: 1) How you would most efficiently **implement** the operation, 2) **Describe** the worst case scenario (e.g. “The worst case occurs when the value you are looking for is not in the list”), if all cases are the same then state: “no worst case”, and 3) What is the worst case **Big-Oh running time** of this scenario. *State any assumptions you make.*

a) (3 points) Push a value onto a stack implemented as an array.

1) Keep an integer top that contains the array index of the top of the stack. Grow the stack from location 0 to max-1, where max is the size of the array. A push operation merely sets array[top] = value, and then increments top (and possibly also a size variable)

2) Unless you are doing something special for the case of a full stack, then all cases are equal. If you have allocated your array on the heap (by storing your stack as a pointer to a contiguous chunk of values that you got off of the heap by calling new) and you want to resize the stack when it gets full, then the worst case would be if you pushed a value onto a stack that was full. In this case you would allocate a new (larger) array off of the heap and copy over the original array values plus the new value into the new array.

3) O(1) if you do not reallocate the array (you either write the new value in or return an error message saying the stack is full). If you copy over the current contents of the stack into a newly allocated stack, then this would have a cost O(n).

b) (3 points) Dequeue a value from a queue implemented as a singly linked list with a dummy header node, where the head of the queue points to the dummy node. [Picture drawn in class]

1) Assuming that you enqueue values at the tail and dequeue them from the head, dequeuing involves returning a pointer to the value pointed to by the dummy node’s next pointer, and then updating the dummy node’s next pointer to point to the following item.

2) All cases are basically the same

3) O(1) as the only operations involved are setting a few pointers, and possibly decrementing a size variable.

c) (3 points) Delete the three largest values found in an unsorted doubly linked list.

There are several ways to do this, but they all end up with the same big-Oh:

1) Initialize three variables (max1, max2, max3) to hold the value of the first item in the list and three pointer variables (max1_ptr, max2_ptr, max3_ptr) to hold pointers to the first node in the list. Search the list from head to tail. Loop thru all values in the list and for each node in the list: if (value > max3) {max3=value, compare to max2, if (value > max2) {max3=max2, max2 = value, compare to max 1, if (value > max1) {max2= max1, max1= value}}} IN ADDITION, keep track of a pointer to the node that contains these values in the max_ptr variables. After you have reached the end of the list, go back and remove the three nodes pointed to by the max_ptr variables.

2) Since you always have to search the entire list, all cases are basically the same.

3) O(n) as you have to search through the entire list to find the 3 max values, then it is just a constant number of pointer updates and delete operations to remove the three values from the list.

Also: do a find max, followed by a delete operation three times. All cases the same. O(n)

10) (7 points) Assume we are using the 32-bit IEEE single precision floating point format as described in class and used in lab. The mantissa has 24 bits including the hidden bit. There is one sign bit and there are eight exponent bits. The exponent is stored in excess 127.

What decimal floating point number is represented by the following 32 bits? SHOW YOUR WORK!

0001 1100 0011 0000 0000 0000 0000 0000

a) Is this a positive or negative number? **positive**

b) What is the exponent (in base 10)?

-71_{10}

c) What is the value of the mantissa (in base 10)

0.375_{10}

(1.375_{10} is also o.k. Technically mantissa refers to the fractional part of a number, but when talking about floating point we typically include the hidden bit in with this. The hidden bit may in fact be part of a fractional value or not, depending on what the value of the exponent is.)

d) What is the total value?

Note: you may leave your answer in the form: $value_{10} * base^{exponent}$
Where you specify value, base and exponent.

$1.375_{10} * 2^{-71}$

11) (16 points) This question tests your understanding of stacks. You must implement a **stack** ADT in C++. The underlying representation of the stack should be a C++ **primitive array**. Your stack should store integers and should handle errors (printing an error message is fine).

You will be graded mostly on the correctness of the ideas of your solution rather than exact C++ syntax, but your solution should be clear. Correct C++ code is the best way to ensure we understand your solution. You may NOT use the STL in any way for this question. You should use the header file provided below.

You should implement all the functions with → in front of them.

```
const int max_size = 100; // max number of elements that
                          // can be stored in the stack

class Stack {
public:
→   Stack(); // constructor
→   void push(int value); // pushes value onto the stack.
→   int pop(); // returns and removes the value on
              // the top of the stack.
→   int top(); // returns the value on the top of
              // the stack without removing it.
→   bool isEmpty(); // returns true if the stack contains
                  // no elements.
→   bool isFull(); // returns true if the stack is full

private:
// Add more data members here if needed.

    int data[max_size];

    int TopOfStack; // index of the next empty position
                  // in the stack that should be filled

};
```

[Extra space for previous problem]

```
Stack::Stack( )
{
    topOfStack = 0;
}

void Stack::push( int x )
{
    if( isFull( ) )
        throw Overflow( );
    data[ topOfStack++ ] = x;
}

int Stack::top( )
{
    if( isEmpty( ) )
        throw Underflow( );
    return data[ topOfStack-1 ];
}

int Stack::pop( )
{
    if( isEmpty( ) )
        throw Underflow( );
    topOfStack--;
    return data[ topOfStack ];
}

bool Stack::isEmpty( )
{
    return topOfStack == 0;
}

bool Stack::isFull( )
{
    return topOfStack == max_size;
}
```

12) (2 points) What is the worst case big-Oh running time of your **pop** method and why?

O(1) – only requires: calling isEmpty (constant time), decrementing a counter, indexing an array – all constant time operations.

13) (2 points) What is the worst case big-Oh running time of your **push** method and why?

O(1) – only requires: calling isFull (constant time), incrementing a counter, indexing an array and storing a value in a location – all constant time operations.

Questions 14 - 16 refer to these declarations:

```
class Node {
    public:
        string Value;
        Node *Next;
};
Node *Temp1;
Node Temp2;
Temp1 = new Node;
```

14) [2 points each] What is the **type** of each the following expressions? Please circle your answer below. Draw pictures if it helps.

a) Temp1.Next

[string] [pointer to string] [Node] [pointer to Node] [pointer to a pointer to a Node] **[illegal]**

b) (*Temp1).Value

[string] [pointer to string] [Node] [pointer to Node] [pointer to a pointer to a Node] [illegal]

c) *(Temp2.Next)

[string] [pointer to string] **[Node]** [pointer to Node] [pointer to a pointer to a Node] [illegal]

d) &Temp2

[string] [pointer to string] [Node] **[pointer to Node]** [pointer to a pointer to a Node] [illegal]

e) Temp1->Value

[string] [pointer to string] [Node] [pointer to Node] [pointer to a pointer to a Node] [illegal]

f) Temp2.Value

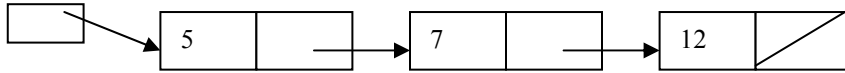
[string] [pointer to string] [Node] [pointer to Node] [pointer to a pointer to a Node] [illegal]

g) &Temp1

[string] [pointer to string] [Node][pointer to Node] **[pointer to a pointer to a Node]** [illegal]

15) [4 points] Assuming that Temp1 points to a list of Nodes that looks like this:

Temp1



Write code that will insert a new Node containing the value 6 into the list after the Node containing 5 and before the Node containing 7. This new node should be allocated on the heap. After inserting the value 6 into the list, your list should appear as in 16) below.

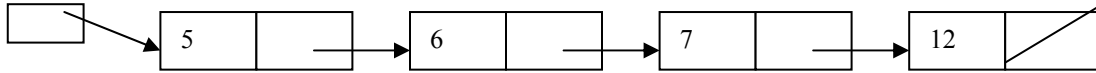
You do not need to write code that searches for the values 5 or 7. Please ask the instructor if you are unsure what is being asked for in this question. Feel free to declare more variables as needed.

```
Node *Temp1;  
// code deleted that builds the list as shown above  
... . . .  
// your code here:
```

```
Node *Temp2 = new Node;  
Temp2-> Value = "6";  
Temp2->Next = Temp1->Next;  
Temp1->Next = Temp2;
```

16) [4 points] Assuming that Temp1 points to a list of Nodes that looks like this:

Temp1



Write code that will remove the Node containing the value 7 from the list and return its memory to the heap. After removing the node, your list should look like the picture shown below. You do not need to write code that searches for the value 7. Please ask the instructor if you are unsure what is being asked for in this question. Feel free to declare more variables as needed.

```
Node *Temp1;  
// code deleted that builds the list as shown above  
... . . . .  
// your code here:
```

```
Node *Temp2 = Temp1->Next->Next;  
Temp1->Next->Next = Temp2->Next;  
OR = Temp1->Next->Next->Next;  
Delete Temp2;  
Temp2 = NULL;
```

After:

Head Temp1

