# **Exam 2 Solution**

*This is a closed book and closed notes test.* You are not allowed to have anything on your desk other than pencil and this exam paper during the test; this includes *calculators* or *electronic assistance* of any kind – ***especially smartphones***.

*You may not leave to go to the restroom.* Please go before the exam starts.

*You may not ask questions.* If something is confusing, write a note beside the question and explain your assumptions.

*You must show all of your work on this exam.* You will not be allowed to turn in additional sheets of paper.

*Read and sign the following statement.*  Failure to sign the statement will result in a **zero** on the exam.

*I have neither given nor received unauthorized assistance on this test. I have notified the proctor of any violations of the above policies.*

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| **Problem** | **Score** |
| 1 | / 25 |
| 2 | / 25 |
| 3 | / 25 |
| 4 | / 25 |
| **Total** | **/ 100** |

*Points divided evenly among parts of a problem unless otherwise specified.*

1. (25 points) Read all of the instructions below carefully. Consider the following class that implements a singly-linked list.

class List {

private:

// ListNode represents each

// node of the list

class ListNode {

public:

int item; // data in the list

ListNode \*next;

ListNode(Item a, ListNode \*n=NULL)

{

item = a;

next=n; // automatically serves as a list tail

}

};

// add head and tail pointer

ListNode \*head;

ListNode \*tail;

public:

List():head(NULL),tail(NULL) {}

// append a to tail of list

void append(int a);

// remove head, and removed item is returned by reference in r

bool removeHead(int &r);

bool empty() const;

**//Room for your own additional declarations:**

};

Implement the following member functions:

|  |  |  |
| --- | --- | --- |
| **Function** | **Function Declaration** | **Description** |
| (a) Copy constructor | List(const List &copy); | List one;  one.append(1);  one.append(2);  // one contains [1,2]  List two = one; // copy one into two  // two contains [1,2] |
| (b) Assignment operator | List& operator=(const List &rhs); | // assume variables one and two from  // prevoius row  List three;  three.append(5);  three = two; // three becomes equal to 2 |

You may assume that any declared functions in the class are implemented, and you may call them from your new functions if that would be helpful. However, anything not declared you will need to declare and implement them. Your implementation must be legal C++ code to earn full credit.

1. Copy Constructor implementation:

|  |
| --- |
| List::List(const List &copy) {  // this is a new list, so initialize head and tail  head = tail = NULL;  // get head of copy  ListNode \* tmp = copy.head;  // iterate over copy and append to this list  while(tmp) {  append(tmp->item);  tmp = tmp->next;  }  } |

(b) Assignment operator implementation.

|  |
| --- |
| List& List::operator=(const List &rhs)  {  // don’t assign if they’re the same!  if(&rhs == this) return \*this;  // free current list  int r;  while(!empty())  removeHead(r);  // this list now empty!  // copy over the rhs, use append to simplify things  ListNode \*tmp = rhs.head;  while(tmp) {  append(tmp->item);  tmp = tmp->next;  }  return \*this;  } |

2. (25 points) Consider the following sequence of operations on a data structure for integers.

* Insert 923
* Insert 714
* Insert 22
* Insert 6735
* Insert 532
* Remove 6735
* Remove 22

1. [13 points] Show the final state of a hash table using linear probing after performing the operations above. Assume the hash table has length 10 and that the hash function is simply index = data % 10. Assume a probeDistance = 5.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Index** | **Value in Table** | **Explanation** | | 0 | empty-since-start |  | | 1 | empty-since-start |  | | 2 | empty-since-removal | 3. Insert 22 here.  7. Remove 22. | | 3 | 923 | 1. Insert 923 here. | | 4 | 714 | 2. Insert 714 here. | | 5 | empty-since-removal | 4. Insert 6735 here.  6. Remove 6735, change to empty-since-removal. | | 6 | 532 | 5. Key is 2, but index=2 is full. Use linear probing and insert here. | | 7 | empty-since-start |  | | 8 | empty-since-start |  | | 9 | empty-since-start |  | |

1. [12 points] Show the final state of an AVL Tree for the same operations. Include your work for partial credit.

|  |
| --- |
| An AVL tree will be left with three nodes after the operations show above. There is only possible configuration in an AVL tree for three nodes:  714 is the root. 532 is the left child. 923 is the right child. Nothing else is possible or it would be imbalanced. |

3. (25 points) Consider the following design requirements for a data structure. For each statement below, explain which data structure you would pick and briefly justify your answer.

1. An arbitrarily large or small number of integers need to be maintained in sorted order. The integers may be inserted in either sorted, partially sorted, or arbitrary order. Irregardless of the order inserted, the data structure maintains their order efficiently.

|  |
| --- |
| An AVL tree provides insertion at O(log n) and ordering in the same cost. It will maintain order efficiently, regardless of how random or ordered they start out with. A BST would not maintain efficient insertion for sorted or near sorted orderings. A linked list would not provide a sorted order efficiently. |

1. A data structure must hold 100 integers. It’s critical that the integers can be accessed by index, for example, get the 5th or 99th integer, efficiently.

|  |
| --- |
| An array. It can be allocated to be size 100 in O(1). It can be indexed in O(1). A hash table would be overkill, but could be used to provide O(1) lookup. Tree data structures would not make sense as they would require O(log n) search even to find the desired index. |

1. A data structure must hold no more than 100 pointers. It must efficiently determine whether or not a given pointer has been inserted.

|  |
| --- |
| A hash table. We just need to determine if a pointer is present, and we know it’s bounded to 100. Hash table can provide O(1) search. All other data structures would be slower. |

1. A data structure must hold pointers to the vertices in a graph. On occasion, you will need iterate over all of the vertices. No particular order of traversal is needed.

|  |
| --- |
| A linked list works well, like an adjancency list. You can insert in O(1) time and traverse in O(N) time. An array is also okay, this is akin to an adjacency matrix. |

1. A data structure must hold the edges in a graph. It must support repeatedly and efficiently removing the edge with the smallest weight from the data structure.

|  |
| --- |
| A min-heap can provide O(log N) removal for each edge. Any ordered tree could also work and provide O(log N) search and removal. A linked list or plain array will incur higher costs to initially sort O(N log N), but they could also be used if the weights do not change. |

4. (25 points) Describe the three cases for removing a node from a binary search tree, and draw an example of each one.

|  |
| --- |
| Case 1: Removing a node with no children. Just disconnect it from its parent by setting the parent’s pointer to NULL. (Include a picture) |
| Case 2: Remove a node with one child. Connect the node’s parent to the child node. (Include a picture) |
| Case 3: Remove an internal node with two children. Find the node’s successor and remove it, recursively. Then, replace the node with its successor. (Include a picture) |

**C++ Keywords**

In common with C:

auto const double float int short struct unsigned  
break continue else for long signed switch void  
case default enum goto register sizeof typedef volatile  
char do extern if return static union while

Unique to C++:

asm dynamic\_cast namespace reinterpret\_cast try  
bool explicit new static\_cast typeid  
catch false operator template typename  
class friend private this using  
const\_cast inline public throw virtual  
delete mutable protected true wchar\_t

Reserved words:

and bitand compl not\_eq or\_eq xor\_eq  
and\_eq bitor not or xor

**ASCII Table**

