Midterm Exam ICS-46, Fall 2016

Signature ____________________________ Name Printed _____________________________

Row ___________ Seat ____________

LEAVE THIS TEST CLOSED, FACE UP, UNTIL YOU ARE INSTRUCTED TO BEGIN

- This is a closed book test. Keep your desk clear (no handouts, notes, books, programs, or calculators).
- You will be provided with an extra sheet that shows all the constructors/methods in the templated classes.
- You should write all your answers clearly on these pages. Make sure that your final answers are easily recognizable. If you need extra space, use the **backs of these pages** for any scratch work. A short, direct answer using the right technical terms is likely to score a higher value than a long rambling answer.
- When you arrive, **pick up the exam and templated classes sheet at the front of the class,** and take your seat. Do not start until I announce when to start (at 2:00pm). When I call time (at 2:50pm), you are to stop writing, stand and **leave by any of the four exits (2 front, 2 back) dropping your exam in the boxes there** (there will also be a box on the stage). Please do this quickly and quietly.
- The number of points for each problem is clearly marked. In total, this test is worth 200 points, so during the 50 minute testing period, you should spend about 1/4 minute per point (4 points in 1 minute). That is about 12 minutes/problem. Not all problems are equally easy. If you get stuck on a problem, move on to another one; return if you have time.
- It is generally better to answer all questions briefly than to answer some completely and leave some blank.
- Hint: sometimes information useful for one problem on the test may be found in other problems on the test.
- When writing answers, it might be useful to **you** to draw pictures of small examples of lists, trees, etc.
- Solutions will be posted in the **Solutions** link on the course web-page soon after everyone takes the exam. I hope that graded exams will be returned next week.

Problem	Points	Score
1	41	41
2	48	89
3	59	148
4	52	200
	200	

Problem	Points	Score

1a. (20 pts). We can use an **ArrayMap** whose **keys** (**std::string**) are each associated with a **value** that is an **ArraySet** of **std::string**. For example, an **ArrayMap** named **test** with four associations might print as:

```
map[a->set[x,y,z], b->set[x,y], c->set[z], d->set[y,z]]
```

Write the **keys_with** function, whose two parameters are (a) first an **ArrayMap** as specified above and (b) and second a **std::string**. The **keys_with** function returns an **ArraySet** of all **key**s whose associated **set** contains the specified **std::string** parameter. Calling **keys_with (test,"y");** would return **set[a,b,d]**: only **c**'s associated **set** does not contain **y**, so it is not included in the returned result.

```
ics::ArraySet<std::string>
keys_with (const ics::ArrayMap<std::string,ics::ArraySet<std::string>>& map, std::string v){
  ics::ArraySet<std::string> answer;
  for (auto kv : map)
    if (kv.second.contains(v))
      answer.insert(kv.first);
 return answer
}

//auto is the type: ics::pair<std::string,ics::ArraySet<std::string>>
```

1b. (12 pts) Assume that we have declared **ics::ArraySet<std::string> words;** write a code fragment that removes those values from **words**, whose **.size** method (callable for **std::string**) call returns a number strictly **>3**.

```
for (auto i = words.begin(); i != words.end(); ++i)
  if (i->size() > 3)
    i.erase(v);
```

1c. (9 pts) For each problem below, fill in the blank with either **DT** (**Data Type)** or **DS (Data Structure)** depending on what you should primarily think about to solve the problem, and then briefly explain your answer.

(a) Design your program: Data Types : use a data type that is correct for the code

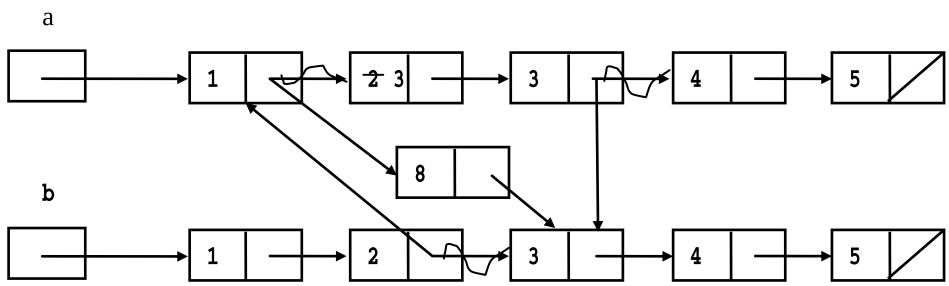
(b) Test your code by running it: Data Structures: run code with any implementation of the data type

(c) Try to improve its efficiency: Data Structures: find a fast/compact implementation of the data type

2. See the class **LN** below. In your answers, you cannot use any other constructors or functions.

```
class LN {
  public:
    LN (int v, LN* n = nullptr) : value(v), next(n){}
    int value;
    LN* next;
};
```

2a. (16 pts) Given the following two linked lists, execute all the statements below, in the specified order, updating the linked lists. **Cross out** ALL values/pointers that are replaced and **Write in** new values/pointers. Use the list updated by one statement as the starting point for the next statement.



1) `a->next->value       = b->next->next->value;`

2) `a->next->next->next  = b->next->next;`

3) `b->next->next        = a;`

4) `b->next->next->next  = new LN(8,a->next->next->next);`

2b. (16 pts) Write an **iterative `append_rear`** function, which appends the specified `int` value (`v`) at the end of a linked list (`l`), which may be an empty list.

```
void append_rear(LN*& l, int v) {
  if (l == nullptr)
    l = new LN(v);
  else {
    for (LN* c = l; /* see body */; c = c->next)
       if (c->next == nullptr) {
          c->next = new LN(v);
          break;
       }
  }
```

```
LN* c = l;
while (c->next != nullptr)
    c = c->next
c->next = new LN(v);
```

2c. (16 pts) Write a **recursive** `copy_deleting` function, which returns a **copy** of the `list` its argument points to, while deallocating all the `LN` objects in that `list`. So, if we executed `x = copy_deleting(x);` then all of the `LN` objects in the `list x` originally pointed to would be deallocated, and `x` would now point to a new `list` that contained all the same values from the original `list x` pointed to. **Hint**: My solution used one local variable, setting/using it once.

```
LN* copy_deleting(LN* l) {
  if (l == nullptr)
    return nullptr;
  else {
    LN* copy = new LN(l->value, copy_deleting(l->next));
    delete l;
    return copy;
  }
```

```
LN* copy_deleting(LN* l) {
  if (l == nullptr)
    return nullptr;
  else {
    LN* tail = copy_deleting(l->next);
    int v = l->value;
    delete l;
    return new LN(v,tail);
  }
```

3a. (26 pts) Write the **complexity class** for each of the following operations. Assume each data structure stores **N** values. **Caution**: some parts of this problem contain information that is **useless/irrelevant** to the question asked.

(a)___O(1)_______ Remove value from position `i` of an **unsorted** array (no need to retain order of elements)

(b)___O(N)______ Remove value from position `i` of a **sorted** array (must retain order of elements)

(c)___O(N)______ Search of an **unsorted** array

(d)_ $O(Log_2N)$____ Search of a **sorted** array

(e)_ $O(Log_2N)$____ Search of a **well balanced** binary search tree

(f)___O(N)______ Search of a **poorly balanced** binary search tree

(g)_ $O(Log_2N)$____ Add a value to a Min-Heap

(h)__O(1)_______ Find (peek at) the minimum value from a Min-Heap

(i)_ $O(Log_2N)$____ Remove the minimum value from a Min-Heap

(j)___O(N)______ Build a heap from N values **offline**

(k) _$O(NLog_2N)$___ Build a heap from N values **online**

(l)__O(N)_______ Reverse a linked list (using the algorithm hand-simulated in class)

(m)_ $O(NLog_2N)$_ Sort a list by copying its values to an array, sorting the array, copying its values back to the list

3b. (16 pts) Assume that function **f** is in the complexity class **O(N** $\times (Log_2N)^2$**)** and that for **N = 10^6 (1,000,000)** the function runs in **8 seconds**.

Write a formula, **T(N)** that computes the approximate time that it takes to run (also in seconds) **f** for any input of size **N**. Show your work/calculations by hand, approximating logarithms, finish/simplify all the arithmetic.

From the formula, **T(N) = c (O(N** $\times (Log_2N)^2$ **)** we have

8 = c$(10^6 \times (Log_2 10^6)^2)$

8 = c (10^6x400)

8 = 4c10^8

therefore,

c = 2x10^{-8}

So, **T(N) = 2x10^{-8}** $\times$ **(N** $\times (Log_2N)^2$**)**

3c. (9 pts) Assume that we have recorded the following data when timing three functions (measured in milliseconds). Based on these, **fill in an estimate** for the complexity class for each method.

N	**Time: Function 1**	**Time: Function 2**	**Time: Function 3**
100	**300**	**20**	**20**
200	**600**	**80**	**22**
400	**1,200**	**320**	**20**
800	**2,400**	**1280**	**22**
1,600	**4,800**	**5,120**	**22**
Complexity Class Estimate	**~doubles: O(N)**	**~ quadruples: $O(N^2)$**	**~ no real increase: O(1)**

3d. (8 pts) Suppose functions **f** and **g** solve the same problem in running times **O(N)** for **f** and **$O(N \log_2 N)$** for **g**. Based on speed alone, when should we use **f** and when should we use **g**? Use one of the words **always**, **sometimes**, or **never** to start your answer, which should include **because** and a short justification.

For solving large problems…

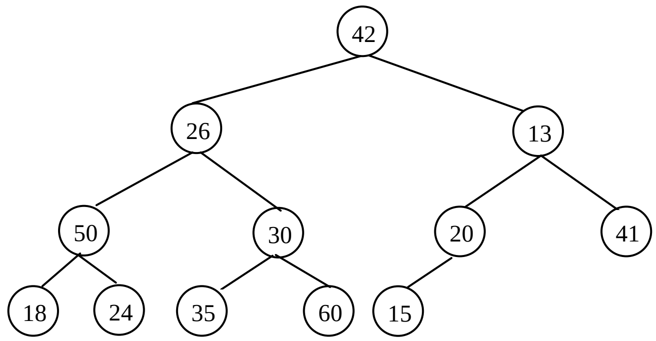
always use **f/never** use **g**, **because** the lower/higher complexity class is guaranteed to be faster/slower beyond some problem size.

For solving small problem…

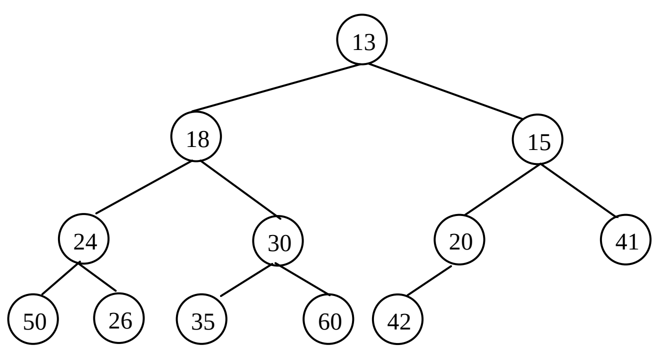
sometimes use **f/sometimes g, because** knowing their complexity class provides no guarantee which is faster for small problem sizes

4a. (8 pts) In the box below, draw a binary tree that **satisfies the structure property** of a **Min-Heap** using the values shown in array below: **values belong in a specific node based on their array index, using the standard encoding of a heap structure as an array**. This binary tree **does not satisfy the order property** for a **Min-Heap** (see part b).

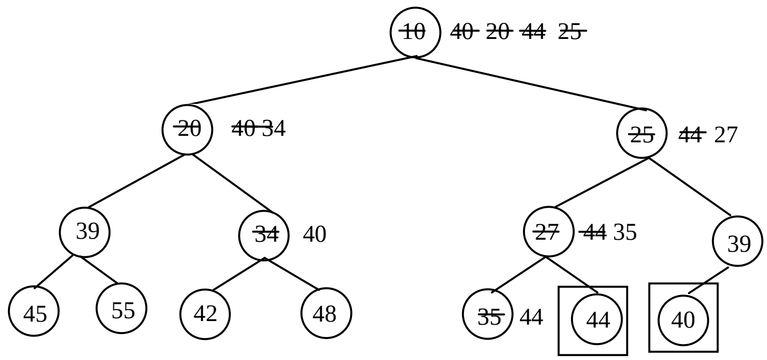
Index	0	1	2	3	4	5	6	7	8	9	10	11
Value	42	26	13	50	30	20	41	18	24	35	60	15



4b. (16 pts) In the box below, draw a binary tree representing a true **Min-Heap** (satisfying both the **order and structure properties**). To construct it, use the standard **offline algorithm** to process the structurally-correct **Min-Heap** from part 4a.



4c. (16 pts) Examine the **Min-Heap** below. Using the standard algorithm, remove the minimum value **twice**: in the tree, draw a box around node(s) that are no longer part of the tree; cross out changed values inside node(s) and write their new value(s) to their right.



4d. (12 pts) Examine the class **`TN`** below.

```
class TN {
  public:
    //..constructors
    int value;
    TN* left;
    TN* right;
}
```

Write the **`height`** function, which computes the height of any node in any binary tree.

```
int height (TN* t) {
  if (t == nullptr)
    return -1;
  else
    return 1 + std::max( height(t->left), height(t->right) )
}
```