

CS 14: Data Structures  
and Algorithms

February 6, 2003

Midterm, Form:

First Name: \_\_\_\_\_

Last Name: \_\_\_\_\_

ID Number: \_\_\_\_\_

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

Take your time and think carefully before you mark your choices. There is plenty of time to think, so don't rush!

Good luck. :)

The questions below are **MULTIPLE CHOICE** questions. Please *circle* one and *only one* of the alternatives provided for each question, unless the question indicates otherwise. Each question is worth **1 point** and **no partial credit is given for any question, unless stated otherwise**. No negative points are given for wrong answers.

1. From **fastest** to **slowest**, the correct order for the following algorithms is:
  - (a) Linear search, Binary search, Bubble Sort, Selection Sort, finding the maximum element of an unsorted array, finding the maximum element of a sorted array.
  - (b) finding the maximum element of an unsorted array, finding the maximum element of a sorted array, Linear search, Binary search, Selection Sort, Bubble Sort.
  - (c) finding the maximum element of a sorted array, Binary search, Linear search, finding the maximum element of an unsorted array, Selection Sort, Bubble Sort.
  - (d) Bubble Sort, Selection Sort, Binary search, Linear search, finding the maximum element of a sorted array, finding the maximum element of an unsorted array.
  - (e) Bubble Sort, Selection Sort, finding the maximum element of an unsorted array, Linear search, Binary search, finding the maximum element of a sorted array.

2. The three classes of ADT operations are:
  - (a) Modifiers, Mutators, and Constructors.
  - (b) Accessors, Observers, and Constructors.
  - (c) Accessors, Mutators, and Destructors.
  - (d) Modifiers, Observers, and Constructors.
  - (e) Modifiers, Observers, and Destructors.
  
3. Consider **OddCounter**, a Counter-like ADT which counts **only odd** numbers (that is, 1, 3, 5, 7, ...). The axiom describing its unique behavior, assuming  $\text{get}(\text{new}()) = 1$ , is:
  - (a)  $\text{get}(\text{inc}(c)) = \text{get}(c) + 2$ .
  - (b)  $\text{get}(\text{inc}(c)) = \text{get}(c) - 2$ .
  - (c)  $\text{get}(\text{inc}(c)) = 2 \text{get}(c) + 1$ .
  - (d)  $\text{get}(\text{inc}(c)) = 2 \text{get}(c) - 1$ .
  - (e) None of the above.
  
4. The correct way to declare a structure to represent a person's family in standard C++ is: [**hint**: two alternatives are syntactically correct but there's still only **one** correct answer]
  - (a) `struct Person { int age; Person *mother; Person *father; }`
  - (b) `struct Person { int age; Person *mother; Person *father; };`
  - (c) `struct Person { int age; Person mother; Person father; }`
  - (d) `struct Person { int age; Person mother; Person father; };`
  - (e) None of the above.

The questions below are **SHORT ANSWER** questions. Please write your answers **only** in the space provided and do so **LEGIBLY**. **WE WILL NOT GIVE YOU CREDIT IF WE HAVE TROUBLE UNDERSTANDING YOUR HANDWRITING, EVEN IF YOUR ANSWER IS CORRECT**. You **must** write your answers using a **permanent pen** or you will **not** receive any credit for them. No negative points are given for wrong answers.

5. [**8 points**] Write down the complexity, in Big-O notation, of the following List operations when applied to a **singly-linked** list:

new:  $\mathcal{O}$ [ \_\_\_\_\_ ]                      addLast:  $\mathcal{O}$ [ \_\_\_\_\_ ]  
size:  $\mathcal{O}$ [ \_\_\_\_\_ ]                      addBefore:  $\mathcal{O}$ [ \_\_\_\_\_ ]  
empty:  $\mathcal{O}$ [ \_\_\_\_\_ ]                      addAfter:  $\mathcal{O}$ [ \_\_\_\_\_ ]  
addFirst:  $\mathcal{O}$ [ \_\_\_\_\_ ]                      remove:  $\mathcal{O}$ [ \_\_\_\_\_ ]

6. [2 points] Using the class `cs14::Array` you used in the lab, write a templated function to determine if an array is sorted in **ascending** order. The function should return `true` if the array passed to it is sorted in **ascending** order, `false` otherwise.
7. [2 points] Starting from an **empty doubly-linked list**, the following operations are performed, in order: `addFirst(A)`, `addFirst(B)`, `addLast(C)`, `addLast(D)`, `insertBefore(2, E)`, `insertAfter(3, F)`, `remove(2)`, where indices start at 0 and A, B, etc, are instances of the `Node` interface. Using **only** the space below (at the end of the question), draw the list that results after those operations. Draw **only** the final result.
8. [2 points] How does the **Selection Sort** algorithm (selecting the **minimum** on each pass) run on the array  $\{7, 2, 3, 6\}$ ? You should use the squares below to trace the algorithm. Start with the elements in the sequence written vertically, with the first (7) on the top-left square and the last (6) on the bottom-left square, then proceed horizontally to the right. You may not need all the squares, so don't feel obliged to use them all.

|   |   |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 0 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 | 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 | 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 | 6 |  |  |  |  |  |  |  |  |  |  |  |  |  |

9. [2 points] How does the **Bubble Sort** algorithm run on the array  $\{7, 2, 3, 6\}$ ? You should use the squares below to trace the algorithm. Start with the elements in the sequence written vertically, with the first (7) on the top-left square and the last (6) on the bottom-left square, then proceed horizontally to the right. You may not need all the squares, so don't feel obliged to use them all.

|   |   |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 0 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 | 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 | 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 | 6 |  |  |  |  |  |  |  |  |  |  |  |  |  |

10. [**3 points**] Define the following complexity-related terms, using **only** the spaces provided. [Note: write **legibly!** We will **not** give you any credit if it takes more than 10 seconds for us to figure out what you've written.]

**best** case complexity:

**worst** case complexity:

**average** case complexity:

# Answer Key for Exam A

Take your time and think carefully before you mark your choices. There is plenty of time to think, so don't rush!

Good luck. :)

The questions below are **MULTIPLE CHOICE** questions. Please *circle one and only one* of the alternatives provided for each question, unless the question indicates otherwise. Each question is worth **1 point** and **no partial credit is given for any question, unless stated otherwise**. No negative points are given for wrong answers.

1. From **fastest** to **slowest**, the correct order for the following algorithms is:
  - (a) Linear search, Binary search, Bubble Sort, Selection Sort, finding the maximum element of an unsorted array, finding the maximum element of a sorted array.
  - (b) finding the maximum element of an unsorted array, finding the maximum element of a sorted array, Linear search, Binary search, Selection Sort, Bubble Sort.
  - (c) finding the maximum element of a sorted array, Binary search, Linear search, finding the maximum element of an unsorted array, Selection Sort, Bubble Sort.
  - (d) Bubble Sort, Selection Sort, Binary search, Linear search, finding the maximum element of a sorted array, finding the maximum element of an unsorted array.
  - (e) Bubble Sort, Selection Sort, finding the maximum element of an unsorted array, Linear search, Binary search, finding the maximum element of a sorted array.
2. The three classes of ADT operations are:
  - (a) Modifiers, Mutators, and Constructors.
  - (b) Accessors, Observers, and Constructors.
  - (c) Accessors, Mutators, and Destructors.
  - (d) Modifiers, Observers, and Constructors.
  - (e) Modifiers, Observers, and Destructors.

3. Consider **OddCounter**, a Counter-like ADT which counts **only odd** numbers (that is, 1, 3, 5, 7, ...). The axiom describing its unique behavior, assuming  $\text{get}(\text{new}()) = 1$ , is:
- (a)  $\text{get}(\text{inc}(c)) = \text{get}(c) + 2.$
  - (b)  $\text{get}(\text{inc}(c)) = \text{get}(c) - 2.$
  - (c)  $\text{get}(\text{inc}(c)) = 2 \text{get}(c) + 1.$
  - (d)  $\text{get}(\text{inc}(c)) = 2 \text{get}(c) - 1.$
  - (e) None of the above.
4. The correct way to declare a structure to represent a person's family in standard C++ is: [**hint**: two alternatives are syntactically correct but there's still only **one** correct answer]
- (a) `struct Person { int age; Person *mother; Person *father; }`
  - (b) `struct Person { int age; Person *mother; Person *father; };`
  - (c) `struct Person { int age; Person mother; Person father; }`
  - (d) `struct Person { int age; Person mother; Person father; };`
  - (e) None of the above.

The questions below are **SHORT ANSWER** questions. Please write your answers **only** in the space provided and do so **LEGIBLY**. **WE WILL NOT GIVE YOU CREDIT IF WE HAVE TROUBLE UNDERSTANDING YOUR HANDWRITING, EVEN IF YOUR ANSWER IS CORRECT.** You **must** write your answers using a **permanent pen** or you will **not** receive any credit for them. No negative points are given for wrong answers.

5. [**8 points**] Write down the complexity, in Big-O notation, of the following List operations when applied to a **singly-linked** list:

**Answer:** In the solution below, `addLast` is  $\mathcal{O}[1]$  if we maintain an extra pointer to track the last element of the list. Otherwise, it's  $\mathcal{O}[n]$ . Similarly, `size` will be  $\mathcal{O}[1]$  if we maintain a count of the number of elements in the list,  $\mathcal{O}[n]$  if we don't.

|                                            |                                               |
|--------------------------------------------|-----------------------------------------------|
| new: $\mathcal{O}[1]$                      | addLast: $\mathcal{O}[1]$ or $\mathcal{O}[n]$ |
| size: $\mathcal{O}[1]$ or $\mathcal{O}[n]$ | addBefore: $\mathcal{O}[n]$                   |
| empty: $\mathcal{O}[1]$                    | addAfter: $\mathcal{O}[1]$                    |
| addFirst: $\mathcal{O}[1]$                 | remove: $\mathcal{O}[n]$                      |

6. [2 points] Using the class `cs14::Array` you used in the lab, write a templated function to determine if an array is sorted in **ascending** order. The function should return **true** if the array passed to it is sorted in **ascending** order, **false** otherwise.

**Answer:**

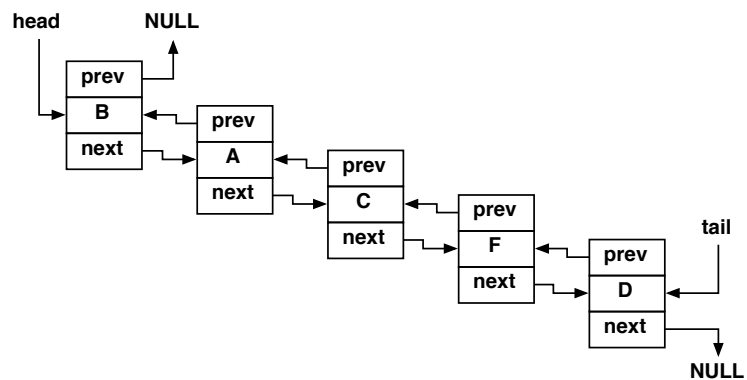
```

template <class T>
bool is_ascending( const Array<T> &a )
{
    for ( int i = 0; i < a.length() - 1; i++)
    {
        if ( a[i] > a[i+1])
            { return false; }
    }
    return true;
}

```

7. [2 points] Starting from an **empty doubly-linked list**, the following operations are performed, in order: `addFirst(A)`, `addFirst(B)`, `addLast(C)`, `addLast(D)`, `insertBefore(2, E)`, `insertAfter(3, F)`, `remove(2)`, where indices start at 0 and A, B, etc, are instances of the `Node` interface. Using **only** the space below (at the end of the question), draw the list that results after those operations. Draw **only** the final result.

**Answer:**



8. [2 points] How does the **Selection Sort** algorithm (selecting the **minimum** on each pass) run on the array  $\{7, 2, 3, 6\}$ ? You should use the squares below to trace the algorithm. Start with the elements in the sequence written vertically, with the first (7) on the top-left square and the last (6) on the bottom-left square, then proceed horizontally to the right. You may not need all the squares, so don't feel obliged to use them all.

**Answer:**

|   |   |   |   |   |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|--|--|--|--|--|--|--|--|--|--|--|
| 0 | 7 | 2 | 2 | 2 |  |  |  |  |  |  |  |  |  |  |  |
| 1 | 2 | 7 | 3 | 3 |  |  |  |  |  |  |  |  |  |  |  |
| 2 | 3 | 3 | 7 | 6 |  |  |  |  |  |  |  |  |  |  |  |
| 3 | 6 | 6 | 6 | 7 |  |  |  |  |  |  |  |  |  |  |  |

9. [2 points] How does the **Bubble Sort** algorithm run on the array  $\{7, 2, 3, 6\}$ ? You should use the squares below to trace the algorithm. Start with the elements in the sequence written vertically, with the first (7) on the top-left square and the last (6) on the bottom-left square, then proceed horizontally to the right. You may not need all the squares, so don't feel obliged to use them all.

**Answer:** The **smart** version of **Bubble Sort** finishes on the next-to-last column, whereas its **dumb** counterpart ends on the last column.

|   |   |   |   |   |   |   |   |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|--|--|--|--|--|--|--|--|
| 0 | 7 | 2 | 2 | 2 | 2 | 2 | 2 |  |  |  |  |  |  |  |  |
| 1 | 2 | 7 | 3 | 3 | 3 | 3 | 3 |  |  |  |  |  |  |  |  |
| 2 | 3 | 3 | 7 | 6 | 6 | 6 | 6 |  |  |  |  |  |  |  |  |
| 3 | 6 | 6 | 6 | 7 | 7 | 7 | 7 |  |  |  |  |  |  |  |  |

10. [3 points] Define the following complexity-related terms, using **only** the spaces provided. [Note: write **legibly!** We will **not** give you any credit if it takes more than 10 seconds for us to figure out what you've written.]

**Answer:**

**best** case complexity: is the running time in  $O()$  notation of an algorithm when it's running on input data of such nature that the algorithm executes the **minimum** number of operations required to complete its execution.

**worst** case complexity: is the running time in  $O()$  notation of an algorithm when it's running on input data of such nature that the algorithm executes the **maximum** number of operations required to complete its execution.

**average** case complexity: is the running time in  $O()$  notation, averaged over multiple random inputs, of an algorithm. It's **always** between best and worst case complexities, but may equal either one or both.