NORTHERN ARIZONA UNIVERSITY
School of Informatics, Computing, and Cyber Systems

CS 249 Data Structures
Review Worksheet

General Instructions

Step 1. Answer the questions below to review for the final exam.

Step 4. Generate a PDF with your solution and submit it as the solution for this assignment.

---

1. Consider the following definition of a linked list node. Write a code snippet that searches the list for the node containing the value 5 and inserts an existing `newNode` after that node. Assume that the variable `Node* current = head;` exists.

```
typedef struct Node
{
    int data;
    struct Node *next;
}
Node;
```

```
// write your code snippet here
while (current != NULL)
{
    if (current->data == 5)
    {
        newNode->next = current->next;
        current->next = newNode;
        break;
    }
    current = current->next;
}
```

2. A stack is implemented using the following structure and operations:

```
typedef struct Stack
{
    int data[MAX];
    int top;
}
Stack;

void push(Stack *s, int value);
int pop(Stack *s);
```

Assume a `Stack s` is initially empty (`top = -1`). The following sequence of operations is executed:

```
push(&s, 1);
push(&s, 2);
push(&s, 3);
pop(&s);
push(&s, 4);
push(&s, 5);
pop(&s);
pop(&s);
push(&s, 6);
```

Fill in the table below with the status of the stack after each operation. Assume each column in the table is the stack at a given moment. Follow the model.

	push(&s,1);	push(&s,2);	push(&s,3);	pop(&s);	push(&s,4);	push(&s,5);	pop(&s);	pop(&s);	push(&s,6);
[4]									
[3]									
[2]									
[1]									
[0]	1								

3. A queue is implemented using the following structure and operations:

```
typedef struct Queue
{
    int data[MAX];
    int front;
    int rear;
}
Queue;

void enqueue(Queue *q, int value);
int dequeue(Queue *q);
```

Assume a `Queue q` is initially empty (`front = rear = -1`). The following sequence of operations is executed:

```
enqueue(&s, 1);
enqueue(&s, 2);
enqueue(&s, 3);
dequeue(&s);
enqueue(&s, 4);
enqueue(&s, 5);
dequeue(&s);
dequeue(&s);
enqueue(&s, 6);
```

Fill in the table below with the status of the queue after each operation. Assume each row in the table is the queue at a given moment. Follow the model.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
enqueue(&q,1);	1						
enqueue(&q,2);							
enqueue(&q,3);							
dequeue(&q);							
enqueue(&q,4);							
enqueue(&q,5);							
dequeue(&q);							
dequeue(&q);							
enqueue(&q,6);							

4. A dequeue is implemented using the following structure and operations:

```
typedef struct Deque
{
    int data[MAX];
    int front;
    int rear;
}
Deque;

void enqueueFront(Deque *d, int value);
void enqueueRear(Deque *d, int value);
int dequeueFront(Deque *d);
int dequeueRear(Deque *d);
```

Assume a `Deque d` is initially empty (`front = rear = -1`). The following sequence of operations is executed:

```
enqueueRear(&d, 1);
enqueueRear(&d, 2);
```

```
enqueueFront(&d, 3);
dequeueRear(&d);
enqueueFront(&d, 4);
dequeueRear(&d);
```

```
enqueueFront(&d, 5);
enqueueRear(&d, 6);
dequeueFront(&d);
dequeueRear(&d);
```

Fill in the table below with the status of the queue after each operation. Assume each row in the table is the queue at a given moment. Follow the model.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
`enqueueRear(&d, 1);`	1						
`enqueueRear(&d, 2);`							
`enqueueFront(&d, 3);`							
`dequeueRear(&d);`							
`enqueueFront(&d, 4);`							
`dequeueRear(&d);`							
`enqueueFront(&d, 5);`							
`enqueueRear(&d, 6);`							
`dequeueFront(&d);`							

5. An iterator is implemented using the following struct. Write one or more statements to fulfill the requests below.

```
typedef struct Iterator
{
    int *array;
    int size;
    int currentIndex;
}
Iterator;
```

a. Declare an iterator variable

```
// write your statement(s) here
Iterator it;
```

b. Dynamically allocate the iterator's array of size 10

```
// write your statement(s) here
it.array = (int*)malloc(10 * sizeof(int));
```

c. Initialize the `size` and `currentIndex` to represent an empty iterator

```
// write your statement(s) here
it.size = 0;
it.currentIndex = 0;
```

d. Write a loop that initializes 5 indexes in the iterator's array with the numbers 1-5. Update the necessary iterator's member(s)

```
// write your statement(s) here
for (int i = 0; i < 5; i++
    {
        it.array[i] = i + 1;
    }
it.size = 5;
```

e. Complete the `hasNext` function to return true if the iterator has a next element or false otherwise

```
bool hasNext(Iterator *it)
{
    // write your statement(s) here
    return it->currentIndez < it->size;

}
```

f. Complete the next function to consume and return the iterator's next element

```
bool next(Iterator *it)
{
    // write your statement(s) here
    if (hasNext(it))
    {
        return it->array[it->currentIndex++];
    }
    else
    {
        return -1;
    }

}
```

6. Given the following unsorted array, write the array after each pass of a **bubble sort** function.

```
int array[10] = {7, 3, 6, 1, 4, 5, 9, 2, 8, 0};
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Pass 1	3	6	1	4	5	7	2	8	0	9
Pass 2	3	1	4	5	6	2	7	0	8	9
Pass 3	1	3	4	5	2	6	0	7	8	9
Pass 4	1	3	4	2	5	0	6	7	8	9
Pass 5	1	3	2	4	0	5	6	7	8	9
Pass 6	1	2	3	0	4	5	6	7	8	9
Pass 7	1	2	0	3	4	5	6	7	8	9
Pass 8	1	0	2	3	4	5	6	7	8	9
Pass 9	0	1	2	3	4	5	6	7	8	9

7. Given the following unsorted array, write the array after each pass of an **insertion sort** function.

```
int array[10] = {7, 3, 6, 1, 4, 5, 9, 2, 8, 0};
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Pass 1	3	7	6	1	4	5	9	2	8	0
Pass 2	3	6	7	1	4	5	9	2	8	0
Pass 3	1	3	6	7	4	5	9	2	8	0
Pass 4	1	3	4	6	7	5	9	2	8	0
Pass 5	1	3	4	5	6	7	9	2	8	0
Pass 6	1	3	4	5	6	7	9	2	8	0
Pass 7	1	2	3	4	5	6	7	9	8	0
Pass 8	1	2	3	4	5	6	7	8	9	0
Pass 9	0	1	2	3	4	5	6	7	8	9

8. Given the following unsorted array, write the array after each pass of a **selection sort** function.

```
int array[10] = {7, 3, 6, 1, 4, 5, 9, 2, 8, 0};
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Pass 1	0	3	6	1	4	5	9	2	8	7
Pass 2	0	1	6	3	4	5	9	2	8	7
Pass 3	0	1	2	3	4	5	9	6	8	7
Pass 4	0	1	2	3	4	5	9	6	8	7
Pass 5	0	1	2	3	4	5	9	6	8	7
Pass 6	0	1	2	3	4	5	6	9	8	7
Pass 7	0	1	2	3	4	5	6	7	8	9
Pass 8	0	1	2	3	4	5	6	7	8	9
Pass 9	0	1	2	3	4	5	6	7	8	9

9. Given the following unsorted array, write the array after each pass of a **quicksort** function, considering that the pivot is always the last element in the subarray and the left side of the pivot is sorted first.

```
int array[10] = {7, 3, 6, 1, 4, 5, 9, 2, 8, 0};
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Pass 1	0	3	6	1	4	5	9	2	8	7
Pass 2	0	1	6	3	4	5	9	2	8	7
Pass 3	0	1	2	3	4	5	9	2	8	7
Pass 4	0	1	2	3	4	5	9	2	8	7
Pass 5	0	1	2	3	4	5	9	2	8	7
Pass 6	0	1	2	3	4	5	9	2	8	7
Pass 7	0	1	2	3	4	5	9	6	8	7
Pass 8	0	1	2	3	4	5	9	6	8	7
Pass 9	0	1	2	3	4	5	9	6	8	7
Pass 10	0	1	2	3	4	5	6	7	8	9

10. Given the following unsorted array, write the array after each pass of a **merge** function in a merge sort algorithm that merges the left subarrays first and the middle element is sorted with the left subarray.

```
int array[10] = {7, 3, 6, 1, 4, 5, 9, 2, 8, 0};
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Pass 1	3	7	1	6	4	5	2	9	0	8
Pass 2	1	3	6	7	2	4	4	9	0	8
Pass 3	1	3	6	7	0	2	2	5	8	9
Pass 4	0	1	2	3	4	5	6	7	8	9
Pass 5	0	1	2	3	4	5	6	7	8	9
Pass 6	0	1	2	3	4	5	6	7	8	9
Pass 7	0	1	2	3	4	5	6	7	8	9
Pass 8	0	1	2	3	4	5	6	7	8	9

11. Given the following array, build a Binary Search Tree (BST) by inserting the elements of the array into the BST in the given order. Draw the tree and list the data in the array below:

```
int array[10] = {7, 3, 6, 1, 4, 5, 9, 2, 8, 0};
```

```
// draw your tree here
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
BST array										

12. Calculate the balance factors for each node in the BST you created in Problem 11

```
// write the balance factors here
7: 2

3: 2
```

```
1: 1

6: 1

9: -1

4: -1

5: 0

2: 0

8: 0

0: 0
```

13. Based on the balance factors you calculated in Problem 12, identify the first unbalanced node and name the necessary rotation to balance the subtree. Draw the tree after performing the rotation on that node and recalculate the balance factors for the tree.

```
Unbalanced node:
Required rotation:

// draw your tree here



// write the balance factors here
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
AVL array										

14. Given the following array, build a **min-heap** tree by inserting the elements of the array into the heap in the given order and heapifying after each insertion. Draw the resulting tree and list the data in the array below:

```
int array[10] = {7, 3, 6, 1, 4, 5, 9, 2, 8, 0};
```

```
// draw your tree here

[0]   [1]   [2]   [3]   [4]   [5]   [6]   [7]   [8]   [9]
0     1     5     2     7     4     6     9     3     8
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Min-heap										

15. Considering the min-heap you created in Problem 14, sort the array in descending order by applying the heapsort strategy. The heapsort algorithm repeatedly extracts the root of the heap and down-heapify the resulting tree until the heap is empty. Draw the tree and the sorted array after each pass

// pass 1

0

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Sorted array										

// pass 2

01

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Sorted array										

// pass 3

012

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Sorted array										

// pass 4
0123

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Sorted array										

// pass 5

01234

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Sorted array										

// pass 6
012345

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Sorted array										

// pass 7
0123456

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Sorted array										

// pass 8

```
01234567
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Sorted array										

```
// pass 9
012345678
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Sorted array										

```
// pass 10
0123456789
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Sorted array										

16. Given a hash function hash(key) = key % 23 and a table of size 23, determine the mapped home bucket for the keys 10, 52, 80, and 124.

```
// write your answers here

Hash key = key % 23:

10 % 23 = 10

52 % 23 = 6

80 % 23 = 11

124 % 23 = 9
```

17. With a hash function hash(key) = key % 11, and a table size of 11, insert the keys 11, 9, 38, 21, 48, 62, 15, 26, 18. Resolve collisions using linear probing. Stop and report it if an overflow happens. Follow the model.

Key	Mapped home bucket	Resolved bucket (after probing)
11	hash(11) = 11 % 11 = 0	-
9	9	9
38	5	5
21	10	10
48	4	4
62	7	7
15	4	6
26	4	8
18	7	8 → 9 → 10 → 0 → 1

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
Hash table	11										

18. Repeat Problem 17 by resolving collisions using quadratic probing. Stop and report it if an overflow happens. Follow the model.

Key	Mapped home bucket	Resolved bucket (after probing)

11	hash(11) = 11 % 11 = 0	-
9		
38		
21		
48		
62		
15		
26		
18		

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
Hash table	11										

19. The following table represents the choices of 6 pet shop customers for 5 different products, where filled cells represent the customer who bought the associated product. For example, according to the table, Alice bought dog food, chew toys, and shampoo.

Customer	Dog Food	Cat Food	Chew Toys	Leash	Shampoo
Alice	■		■		■
Bob	■			■	
Charlie		■	■		
Diana		■	■		■
Eric	■		■	■	
Fiona		■	■		■

Draw an undirect graph to represent the relationship between **products**. In this graph, each vertex represents a product and an edge between two products means that at least one customer purchased them together. Add a weight to the edges to indicate how often the products are bought together.

```
// draw or paste an image of your graph here
```

20. Represent the graph from Problem 19 using an adjacency matrix. Use the table below to facilitate the representation. Label rows and columns. Empty cells (no edge between the nodes) are set with 0.

■					
	■				
		■			
			■		
				■	
					■

21. Assume the adjacency matrix is stored in the memory as a 2D array of short integers (2 bytes each). How much memory in bytes is necessary to store the matrix?

```
// write your answer here
50 bytes.
```