CS 261-020 Exam I Winter 2022

Part I: True (T) / False (F), put T/F on the answer sheet (30 pts, 2 pts each)

1. Like in C++, C also supports pass by reference.

2. Data structures are general-purpose mechanisms for storing, organizing, and managing data within a program.

3. Binary search can be used on an array and a singly linked list, and both would have O(log n) runtime complexity.

4. The maximum number of comparisons required to find a value in a collection of 20 items using a binary search is 4.

5. Since an array provides direct (random) access, the worst-case runtime complexity of array insertion is constant.

6. The action of hiding the internal details of a data type is called encapsulation.

7. Different kinds of data collection (stack, queue, etc.) can share the same type of iterator.

8. The **`next()`** function in iterator indicates whether or not there is another element afterwards.

9. If data in a program is frequently changing, it is more efficient to run a general-purpose sorting algorithm over binary search after each insertion to maintain an ordered array.

10. When allocating 100 integers on the heap using the following statement, both the pointer variable **`array`** and where **`array`** points to are on the heap.
    ```
    int * array = malloc(100 * sizeof(int));
    ```

11. The operation to remove the top element in a stack is called pull.

12. When implementing a deque using linked list with sentinel, both insertions functions can use the same mechanics by calling a helper function.

13. When access and add elements, the stack uses "first in first out (FIFO)" rule, while the queue uses "last in first out (LIFO)" rule.

14. You can directly access the **n**th node of a linked list.

15. In a program that often needs to insert an element into the data collection, we'd better to choose a linked list over an array.

Part II: Multiple Choices. Put your answers on the answer sheet (42 pts, 2 pts each)

1. Which of the following data structures is non-linear?
   A. Stacks
   B. Deques
   C. Circular linked lists
   D. Trees

2. A linear list in which the last node points to the first node is called _____.
   A. singly linked list
   B. doubly linked list
   C. circular linked list
   D. None of the above

3. What's the output of the following program?

```
int main() {
      char* name = "Roger";
      name[0] = 'T';
      printf("%s", name);
      return 0;
}
```

   A. Runtime error.
   B. Compiling error.
   C. **Toger**.
   D. The memory address of **name**

4. You are experimenting with a function which has average-case runtime complexity **$O(n^3)$**, and you've found that, on average, this function runs in 10ms on the input size of 10,000. How long, then, would you expect this function to run on the input size of 20,000?
   A. 40 ms
   B. 80 ms
   C. 20 ms
   D. 90 ms

5. What is the runtime complexity of the following snippet of code?

```
double v = 0;
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++)
         v += i * j;
}
for (int i = 0; i < n; i++)
    v -= i * i;
```

   A. O(n)
   B. $O(n^3)$
   C. $O(n^2+n)$
   D. $O(n^2)$

6. What argument do we pass into **`malloc()`** when allocating memory on the heap?
   A. The data type of which we want to allocate
   B. The number of bytes
   C. Nothing
   D. The number of elements

7. In a linked list, the pointer variable **`struct node* head`**
   A. points to the first node in the list
   B. is the first node in the list
   C. is undefined
   D. is always NULL

8. At maximum, how many characters can we store in the C string **`str`** without causing any potential issues later?
```
char* str = malloc (9 * sizeof(char));
```
   A. 9
   B. 10
   C. 8
   D. 7

9. The void pointer can point to which of the following data type?
   A. **`int`**
   B. **`struct some_struct`**
   C. **`double`**
   D. all of the above

10. In C, we use function pointers to avoid code redundancy. What does the function pointer store?
    A. The entire function body
    B. The return value of a function
    C. The memory address of a function
    D. The memory address of the return value of a function

Questions 11 – 16 are based on the following code:

```
//dynamic array structure
struct dynarray{
    ___①___ data;
    int size;
    int capacity;
};

struct dynarray* dyn_create(){
   //allocates and initialize an empty dynamic array and
   //return a pointer to it
}
```

```
void dyn_free(struct dynarray* da){
    //free the memory associated with a dynamic array
    _____②_____
}

void dyn_insert(struct dynarray *da, void* val){
    //insert a new value to a given dynamic array
}

void dyn_print(struct dynarray *da, void(*prt)(int, void**)){
    prt(da->size, da->data);
}

void dyn_sort(struct dynarray* da, ______③______){
    for (int i = 0; i < da->size; i++){
        for (int j = i+1; j < da->size; j++){
            if (_____④_____){
                //swap the values…
            }
        }
    }
}


//helper functions
int compare_ints(void* x, void* y){
    int* a = x, *b = y;
    return *a > *b ? 1 : 0;
}

void print_ints(int size, void** data){
    //print every element in data
}


int main(){
    int n = 3;
    struct dynarray *da = dyn_create();
    int* temp = malloc(n * sizeof(int));

    for (int i = 0; i < n; i++){
        temp[i] = n-i;
        dyn_insert(da, _____⑤_____);
    }
```

```
    //print before sort()
    dyn_print(da, print_ints);

    _________⑥_________

    //print after sort()
    dyn_print(da, print_ints);

    //free memory
    dyn_free(da);
    free(temp);

    return 0;
}
```

11. Assuming everything else works correctly, what should be filled at ① so **`data`** can be used as a pointer to a dynamic array of any data type?
    A. **`void`**
    B. **`void*`**
    C. **`void**`**
    D. **`int*`**

12. Assuming everything else works correctly, what should be filled at ② so **`dyn_free()`** would properly free the memory associated with the dynamic array?
    A.
    ```
    free(da);
    ```
    B.
    ```
    free(da->data);
    free(da);
    ```
    C.
    ```
    free(da);
    free(da->data);
    ```
    D.
    ```
    for(int i = 0; i < da->size; i++)
        free(da->data[i]);
    free(da->data);
    free(da);
    ```

13. Assuming everything else works correctly, what should be filled at ③ so that **`dyn_sort()`** is able to compare the values in the given dynamic array, i.e., by using **`compare_ints()`**?
    A. **`int cmp (void*, void*)`**
    B. **`int (*cmp (void*, void*))`**
    C. **`int (*cmp) (void*, void*)`**
    D. **`void* (*cmp) (void*, void*)`**

14. Assuming everything else works correctly, what should be filled at ④ so that we can use the function pointer **cmp** to compare the two elements in the array, **da->data[i]** and **da->data[j]**?
    A. **cmp(da->data[i], da->data[j])**
    B. **(*cmp) (da->data[i], da->data[j])**
    C. **cmp(da->data[i]), cmp(da->data[j])**
    D. **(*cmp) (da->data[i]), (*cmp) (da->data[j])**

15. Assuming everything else works correctly, what should be filled at ⑤ so that we can insert **temp[i]** into the dynamic array **da** correctly?
    A. **(void*) temp[i]**
    B. **(void*)&temp[i]**
    C. **(void*) temp**
    D. **(void*) *temp[i]**

16. Assuming everything else works correctly, what should be filled at ⑥ so that we can sort the elements in the dynamic array properly?
    A. **dyn_sort(da, compare_ints);**
    B. **dyn_sort(da, cmp);**
    C. **dyn_sort(da, *compare_ints);**
    D. **dyn_sort(da, *cmp);**

Questions 17 – 21 are based on the following code:

Suppose we are using a dynamic array with circular buffer that reindexes after resizing to implement a queue, **q**, which is set up to hold integer values. Assuming the initial size and capacity of **q** are 0 and 2, respectively, and the capacity is doubled when resizing.

After executing following sequence of operations …

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

17. What is the size of **q**?
    A. 6
    B. 5
    C. 4
    D. 3

18. What is the capacity of **`q`**?
    A. 1
    B. 2
    C. 4
    D. 8

19. What is the physical index of **`start`** (first element in the queue)?
    A. 0
    B. 1
    C. 2
    D. 3

20. What is the logical index of the element of value 6?
    A. 0
    B. 1
    C. 2
    D. 3

21. What is the physical index of the element of value 5?
    A. 0
    B. 1
    C. 2
    D. 3

Part III: Matching and short answers. Put your answers on the answer sheet (28 pts)

1. (14 pts, 2 pts each) Based on the following function body implementation, select the corresponding function names (A-I) for ① to ⑦.

   Possible options: (Note: you have more options than you need)

   A. **`list_create`** // allocate and initialize a list
   B. **`list_free`** // free the memory associated with the list
   C. **`list_addFront`** // add a node to the front of the list
   D. **`list_addBack`** // add a node to the back of the list
   E. **`list_removeFront`** // remove the first node from the list
   F. **`list_removeBack`** // remove the last node from the list
   G. **`list_print`** // print each element of the list
   H. **`list_position`** // return the index of the first instance of a given value
   I. **`list_reverse`** // reverse the order of the nodes in a list

   ①: ②:

   ③: ④:

   ⑤: ⑥: ⑦:

```
struct node{
     int val;
     struct node* next;
};

struct list{
     struct node* head;
};

struct list* ____①_____(){
     struct list* ll = malloc(sizeof(struct list));
     ll->head = NULL;
     return ll;
}

void ____②____ (struct list* ll, int val){
     struct node* temp = malloc(sizeof(struct node));
     temp->val = val;
     temp->next = ll->head;
     ll->head = temp;
}

void ____③____ (struct list* ll){
     if (ll->head == NULL)
          return;
     struct node* temp = ll->head->next;
     free(ll->head);
     ll->head = temp;
}

void ____④____ (struct list* ll, int val){
     struct node* temp;
     temp = malloc(sizeof(struct node));
     temp->val = val;
     temp->next = NULL;

     if (ll->head == NULL){
          ll->head = temp;
          return;
     }
     struct node* curr = ll->head;
     while(curr->next != NULL){
          curr = curr->next;
     }
     curr->next = temp;
}
```

```
void ____⑤____ (struct list* ll){
     if (ll->head == NULL){
          return;
     }
     if (ll->head->next == NULL){
          free(ll->head);
          ll->head = NULL;
          return;
     }
     struct node* curr = ll->head;
     while(curr->next->next != NULL){
          curr = curr->next;
     }
     free(curr->next);
     curr->next = NULL;
}

void ____⑥____ (struct list* ll){
     struct node* temp;
     while(ll->head != NULL){
          temp = ll->head;
          ll->head = ll->head->next;
          free(temp);
     }
     free(ll);
}

void ____⑦____ (struct list* ll){
     struct node* temp = ll->head;
     while(temp != NULL){
          printf("%d\t", temp->val);
          temp = temp->next;
     }
     printf("\n");
}
```

2. (4 pts) Given two empty queues, q1 and q2, what will be printed by the following sequence of operations? (Assuming the queues are set up to hold integer values).

```
enqueue(q1, 1);
enqueue(q1, 2);
enqueue(q1, 3);
enqueue(q1, 4);
enqueue(q2, dequeue(q1));
enqueue(q2, dequeue(q1));
enqueue(q2, dequeue(q1));
enqueue(q2, dequeue(q1));
printf("%d", dequeue(q2)); // 1st print
printf("%d", dequeue(q2)); // 2nd print
printf("%d", dequeue(q2)); // 3rd print
printf("%d", dequeue(q2)); // 4th print
```

1st print: _____
2nd print: _____
3rd print: _____
4th print: _____

3. (6 pts) Given an empty stack, **s**, what will be printed by the following sequence of operations? (Assuming the stack is set up to hold integer values).

```
push(s, 1);
push(s, 2);
push(s, 3);
push(s, 4);
printf("%d", pop(s));  //1st print
printf("%d", pop(s));  //2nd print
push(s, 5);
push(s, 6);
push(s, 7);
printf("%d", pop(s));  // 3rd print
printf("%d", pop(s));  // 4th print
printf("%d", pop(s));  // 5th print
printf("%d", pop(s));  // 6th print
```

1st print: _____
2nd print: _____
3rd print: _____
4th print: _____
5th print: _____
6th print: _____

4. (4 pts) Suppose the stack **s** in the previous question is implemented using a dynamic array, and its initial size and capacity are 0 and 2, respectively. What are the values of size and capacity after executing the operations in the previous question? (Assuming the capacity is doubled when resizing)
size: capacity:

Extra Credit: Put your answers on the answer sheet

1. (3 pts) What's the output of the following program?

```
int main() {
      int a = 128;
      void* ptr = &a;
      printf("%d", *ptr);
      return 0;
}
```

   A. Runtime Error.
   B. Compiling error.
   C. 128.
   D. The memory address of **a**

2. (5 pts): What's the output of the following program, and why?

```
int main() {
      int a = 128;
      char *ptr;
      ptr = (char *) &a;
      printf("%d", *ptr);
      return 0;
}
```