List, Set, Map, Stack, Queue and Tree

A brief recap

Data Structure	Definition	Ordered?	Allows Duplicates?	Allows Nulls?	Add/Remove	Examples
List	A collection that maintains the order of its elements	Yes	Yes	Yes	Add/remove at specified index or end	ArrayList, LinkedList
Set	A collection that does not allow duplicates	No	No	Yes (only one null value allowed)	Add/remove elements	HashSet, TreeSet
Map	A collection that stores key-value pairs, with no duplicate keys	No	Values only	Yes (only one null key allowed)	Add/remove key-value pairs	HashMap, TreeMap
Stack	A collection that implements a last-in-first-out (LIFO) policy	Yes	Yes	Yes	Push/pop elements	Stack
Queue	A collection that implements a first-in-first-out (FIFO) policy	Yes	Yes	Yes	Add to end/remove from front	LinkedList, PriorityQueue
Tree	A hierarchical data structure that stores elements in a tree-like structure	Yes	Yes	Yes	Add/remove nodes	BinarySearchTree, RedBlackTree

- LinkedList is yet another implementation of the List interface (ArrayList and Vector were too).
- The main difference between LinkedList and ArrayList is that the latter is implemented using arrays (a contiguous memory block) while the former is not. Each element of a LinkedList contains references to the next and previous elements in the list (this variant is sometimes called doubly-linked list) and each element may reside in arbitrary locations in memory.
- We prefer ArrayList when we want to keep items in an order and want to access them randomly using their indices. We use LinkedList when we want to keep items in an order and add or delete elements at random positions. Deletion or insertion requires shifting of many elements (comparable to the size of the whole list) when we use ArrayList. LinkedList has the big advantage of flexible structure for these operations and it doesn't require any shifting.

HashMap:

- Conceptually, HashMap is a generalization of ArrayList in the sense that ArrayList of type T can be thought of a map from nonnegative integer values (indices) to values of type T.

- Maps make it possible to use data types other than integers to be used as indices (in the map terminology they are called keys) to access the values.
- We use HashMap when we need to store an association between two kinds of entities and want to access one of them by querying the other.

HashSet:

- We use HashSet when we are not interested in any kind of ordering for our collection but we want our elements to be unique.
- The main operation supported by sets is the check of membership, whether an element is a member of the set or not.

Stack:

- Conceptually it is like a "stack of books", which you can add or remove "books" but only from the top. It is a LIFO (last in first out) structure.
- Main operations associated with a stack are adding a new element on top of it (aka push), removing an element from the top (aka pop), peeking the element on the top, and checking the size.
- In Java Collections we have a Stack class which relies on (extends) Vector class we have learned before.

Queue:

- Conceptually it resembles a real-life queue where people leave it from front (after being served let's say) and new people join from the behind. It is a FIFO (first in first out) structure.
- Main operations associated with a queue are adding a new element to queue (aka enqueue), remove an element from the queue (aka dequeue), peeking the front element of the queue and checking the size.
- We'll use LinkedList class of Java for our queue needs.

Critical Thinking

1. "We have a list consisting of all numbers from 1 to 100. Each time; we remove two numbers randomly from the list, add them up, subtract 2 from the sum and add the result back in the list. We continue this process until there is exactly one number left in the list. What i that last number?"

Suppose that you want to solve this problem by simulating the whole process using your computer. You decided to use Java language. Which of the following would be more appropriate to store the numbers?

a) ArrayList

b) LinkedList

c) HashSet

d) HashMap

solution: B

---

2. "Write a function which takes a string representing an English sentence and returns the number of occurrences of each word in the string. You can assume that all the words are separated with a single space and there is no punctuation mark."

Which of the following is more appropriate to use when you want to solve this question?

a) HashSet

b) LinkedList

c) HashMap

d) ArrayList

solution: C

---

3. What is wrong with the following statement?

```
HashMap<String> map = new HashMap<>();
```

a) HashMaps cannot be created empty.

b) The right-hand side of assignment must be `new HashMap<String>();`

c) HashMaps must be specified with two data types, one for key and the other for value.

d) You have to specify an initial size while creating a HashMap.

solution: C

---

4. The following piece of code is supposed to take integers from user and store them in an ArrayList until the user enters 0. Which one should be replaced with question marks?

```
Scanner sc = new Scanner(System.in);
ArrayList<Integer> list = new ArrayList<>();
int num;
while(??)
    list.add(num);
```

a) `(num = sc.nextInt()) != 0`

b) `num = (sc.nextInt() != 0)`

c) `true`

d) `num != 0`

solution: A

---

5. The following function is supposed to convert an array into a set, to get rid of any repetitions. What should come in place of the question marks?

```
static HashSet<String> setOf(String[] arr){
    HashSet<String> set = new HashSet<>();
    for(String s:??) {
        set.add(s);
    }
    return set;
}
```

   a) `set`

   b) `arr`

   c) `arr[i]`

   d) `set[i]`

solution: B

---

6. Consider the following function which is supposed to do something similar to what `get()` method of a `HashMap<String, Boolean>` object does. What is the main difference between the original method and this replica?

```
static Boolean myMapGet(String s, String[] strArr, Boolean[] boolArr) {
    int i;
    for(i=0; i<strArr.length; i++)
        if(strArr[i].equals(s))
            break;
    // nothing found
    if(i==strArr.length)
        return null;
    else
        return boolArr[i];
}
```

   a) The original get() method does not return null in any case.

   b) This method returns an instance of the wrapper class Boolean while the original get() method would return a primitive boolean value.

   c) Everything is different. It is not possible to write a method which maps different data types without using a real HashMap instance.

   d) This method searches for the key by traversing all keys and returns the corresponding value, while the original get() method uses a completely different technique. There is a significant performance difference between the two.

solution: D

---

7. What is wrong with the following program?

```
public static void main(String[] args) {
    Stack<String> stack = new Stack<>();
    stack.push("item");
    stack.pop("item");
}
```

a) A stack must be created with an initial capacity.

b) Identifier names cannot be the same as their type.

c) Pop method does not take any parameter.

d) Push method needs one more parameter.

solution: C

---

8. What is wrong with the following program?

```
public static void main(String[] args) {
    Stack<Character> st = new Stack<>();
    st.push(null);
    st.pop();
    st.pop();
}
```

a) You cannot push null values to a character stack.

b) You must provide a parameter for pop method, because you must specify what you want to remove.

c) Push method needs to know position information, one more argument is needed.

d) You cannot pop an element from an empty stack.

solution: D

---

9. What does the following program do?

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    Stack<Character> st = new Stack<>();
    char c;
    while((c = sc.next().charAt(0)) != 'q')
        st.push(c);
    while(!st.empty())
        System.out.println(st.pop());
}
```

a) Stores characters until a 'q' is entered and prints stored characters to the screen in reverse order.

b) Stores characters until a 'q' is entered and prints the second last (right before 'q') character to the screen.

c) Stores characters until a 'q' is entered and prints stored characters to the screen in the order they were given.

d) Waits for user to enter a 'q' character and begins storing the characters thereafter.

solution: A

---

10. Do you think the following program will terminate? Assume that your computer has infinite memory.

```
public static void main(String[] args) {
    LinkedList<Double> queue = new LinkedList<>();
    while(true)
        if(Math.random() < ??)
            queue.removeFirst();
        else
            queue.addLast(Math.random());
}
```

a) Yes

b) No, since the condition of while loop is always true.

c) Depends on the value that replaces the question marks. If it is larger than 0.5, the program will halt eventually almost for sure.

d) Depends on the value that replaces the question marks. If it is less than 0.5, the program will halt eventually almost for sure.

solution: C

---

11. The following program is supposed to implement a simple calculator. What should be written in place of the comment?

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    Stack<Integer> st = new Stack<>();
    while(true) {
        String token = sc.next();
        switch(token) {
        case "+":
            st.push(st.pop() + st.pop());
            break;
        case "=":
            System.out.println(st.peek());
            break;
        default:
            // here
        }
    }
}
```

a) `break;`

b) `st.push(Integer.*valueOf*(token));`

c) `st.push(token);`

d) `st.pop();`

solution: B

Practice

1. Write a program which solves the problem quoted below.

We have a list consisting of all numbers from 1 to 100. Each time; we remove two numbers randomly from the list, add them up, subtract 2 from the sum and add the result back in the list. We continue this process until there is exactly one number left in the list. What i that last number?

solution:

```
public static void main(String[] args) {
    LinkedList<Integer> list = new LinkedList<>();
    Random r = new Random();
    // initialize
    for(int i=1; i<=100; i++)
        list.add(i);
    while(list.size() > 1) {
        // draw first
        int randX1 = r.nextInt(list.size());
        int num1 = list.remove(randX1);
        // draw second
        int randX2 = r.nextInt(list.size());
        int num2 = list.remove(randX2);
        // add their sum - 2
        list.add(num1+num2-2);
    }
    System.out.println(list);
}
```

---

2. Write a function which takes a string representing an English sentence and returns the number of occurrences of each word in the string. You can assume that all the words are separated with a single space and there is no punctuation mark.

Main method:

```
public static void main(String[] args) {
    String sentence = "I am a student I am a student of computer science I am a student of computer science of t
    HashMap<String, Integer> counts = countWords(sentence);
    System.out.println(counts);
}
```

solution:

```
static HashMap<String, Integer> countWords(String sentence){
    HashMap<String, Integer> counts = new HashMap<>();
    for(String s: sentence.split(" ")) {
        if(counts.containsKey(s))
            counts.put(s, counts.get(s) + 1);
        else
            counts.put(s, 1);
    }
    return counts;
}
```

---

3. Write a function which takes a `HashMap<String, Integer>` `map1` and a `HashMap<Integer, String>` `map2` as parameters and returns their composite mapping, which is of type `HashMap<String, String>`. The composition operation is the same with function composition in mathematics.

Main method:

```
public static void main(String[] args) {
    HashMap<String, Integer> map1 = new HashMap<>();
```

```
    map1.put("a", 1);
    map1.put("b", 2);
    map1.put("c", 3);
    HashMap<Integer, String> map2 = new HashMap<>();
    map2.put(1, "one");
    map2.put(2, "two");
    map2.put(3, "three");
    HashMap<String, String> res = compose(map1, map2);
    System.out.println(res);
}
```

solution:

```
static HashMap<String, String> compose(HashMap<String, Integer> map1, HashMap<Integer, String> map2) {
    HashMap<String, String> res = new HashMap<>();
    for(String key: map1.keySet()) {
        Integer n = map1.get(key);
        if(map2.containsKey(n))
            res.put(key, map2.get(n));
    }
    return res;
}
```

---

4. You are given a mapping which maps a child to his/her dad. Each person is uniquely represented with his/her ID number, which is just some string of characters. Your task is to find out how many children each dad has.

In Java terms, you should write a function which takes a parameter of type `HashMap<String, String>` and returns an object of type `HashMap<String, Integer>`.

Main method:

```
public static void main(String[] args) {
    HashMap<String, String> dadMap = new HashMap<>();
    dadMap.put("child1", "dad1");
    dadMap.put("child2", "dad1");
    dadMap.put("child3", "dad2");
    dadMap.put("child4", "dad2");
    dadMap.put("child5", "dad2");
    dadMap.put("child6", "dad3");
    dadMap.put("child7", "dad3");
    dadMap.put("child8", "dad3");
    dadMap.put("child9", "dad3");
    System.out.println(childrenCount(dadMap));
}
```

solution:

```
static HashMap<String, Integer> childrenCount(HashMap<String, String> dadMap) {
    HashMap<String, Integer> counts = new HashMap<>();
    for(String child: dadMap.keySet()) {
        String dad = dadMap.get(child);
        if(counts.containsKey(dad)) // not the first child
            counts.put(dad, counts.get(dad) + 1);
        else // the first child
            counts.put(dad, 1);
    }
    return counts;
}
```

---

5. Imagine that you work in a hypothetical village and your job is to arrange marriages of young people. You get two mappings from the delegates. One of them maps boys to girls, showing which boy wants to marry which girl, and the other is vice versa. Your task is to find out mutual mappings (A loves B and B loves A), so you can declare their marriage.

Assume that each person is represented with names in the lists and no two people have the same name in this village.

In Java terms, you should write a function which takes two parameters of type `HashMap<String, String>` and returns an object of type `HashMap<String, String>`.

Main method:

```
public static void main(String[] args) {
    HashMap<String, String> boyzList = new HashMap<>();
    boyzList.put("boy1", "girl1");
    boyzList.put("boy2", "girl2");
    boyzList.put("boy3", "girl3");
    HashMap<String, String> girlzList = new HashMap<>();
    girlzList.put("girl1", "boy1");
    girlzList.put("girl2", "boy3");
    System.out.println(marry(boyzList, girlzList));
}
```

solution method:

```
static HashMap<String, String> marry(HashMap<String, String> boyzList,HashMap<String, String> girlzList) {
    HashMap<String, String> marriages = new HashMap<>();
    for(String boy: boyzList.keySet()) {
        String girl = boyzList.get(boy);
        if(girlzList.containsKey(girl) && girlzList.get(girl).equals(boy))
            marriages.put(boy, girl);
        else;
        // boy listens a love song
    }
    return marriages;
}
```

6. Consider a set of matryushka dolls each of which has a unique name. Assume that each doll is either small or medium or large in size. You are given two mappings. The first mapping maps big dolls to medium dolls, indicating which doll contains which. The second mapping maps medium dolls to small dolls, with the same meaning as that of first. Your task is to come up with a map from big dolls to small dolls showing containment information.

Note that there may be a big doll containing a medium doll which does not contain any small doll. For such cases you should map big doll to void.

In Java terms, you should write a function which takes two parameters of type `HashMap<String, String>` and returns an object of type `HashMap<String, String>`.

Main method:

```
public static void main(String[] args) {
    // BtM: big to medium
    HashMap<String, String> BtM = new HashMap<>();
    BtM.put("big1", "medium1");
    BtM.put("big2", "medium2");
    BtM.put("big3", "medium3");
    // MtS: medium to small
    HashMap<String, String> MtS = new HashMap<>();
    MtS.put("medium1", "small1");
    MtS.put("medium2", "small2");
```

```
    MtS.put("medium3", "small3");
    System.out.println(matryushka(BtM, MtS));
}
```

solution:

```
static HashMap<String, String> matryushka(HashMap<String, String> BtM, HashMap<String, String> MtS) {
    // BtS: big to small
    HashMap<String, String> BtS = new HashMap<>();
    for(String bigDoll: BtM.keySet()) {
        String mediumDoll = BtM.get(bigDoll);
        if(MtS.containsKey(mediumDoll))
            BtS.put(bigDoll, MtS.get(mediumDoll));
        else
            BtS.put(bigDoll, null);
    }
    return BtS;
}
```

---

7. You and your n friends decided to play a deadly game. You bought n bombs. Each bomb has a little machinery connected to a button such that when the button is pressed the bomb explodes with probability p.

You form a circle to play this game. The game is played in turns, in each turn the current player presses the button and if he stays alive he hands the bomb to the friend on his left or right, with equal probability. When a friend dies, a random friend from the remaining pool takes a new bomb and the game continues until one person is left, which is the winner of the game. Game starts with a random friend.

Assume that a bomb kills only the person who holds it.

Write code to simulate this game.

Main method:

```
public static void main(String[] args) {
    LinkedList<String> friends = new LinkedList<>();
    friends.add("A");
    friends.add("B");
    friends.add("C");
    friends.add("D");
    friends.add("E");
    System.out.println(fancySuicide(friends, .5));
}
```

solution:

```
// p is the probability of explosion
static boolean prob(double p) {
    return Math.random() < p;
}

static String fancySuicide(LinkedList<String> friends, double p) {
    Random r = new Random();
    int turn = r.nextInt(friends.size());
    while(friends.size() > 1) {
        if(prob(p)) {
            friends.remove(turn);
            turn = r.nextInt(friends.size());
        }
        else
            if(prob(.5))
                turn = (turn + 1) % friends.size();
```

```
            else// ensure that the result of % is non-negative
                turn = (turn + friends.size() - 1) %
        friends.size();
    }
    return friends.getFirst();
}
```

---

8. Write a program which provides a command-line interface to a stack of strings. Your program should push a string to a stack when user enters "push string". Likewise it should pop a string from the stack and prints it to the screen when user enters "pop". The program should warn the user when she tries to pop from an empty stack. Lastly, when user enters "quit", the program should report the number of items remaining in the stack and terminate.

solution:

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    Stack<String> st = new Stack<>();
    String in;
    while(!(in = sc.next()).equals("quit"))
        if(in.equals("push"))
            st.push(sc.next());
        else if(in.equals("pop"))
            if(st.empty())
                System.out.println("stack is empty!");
            else
                System.out.println(st.pop());
    System.out.println(st.size() + " elements left.");
}
```

---

9. Implement the same program for a queue structure, use addLast and removeFirst methods of LinkedList class.

solution:

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    LinkedList<String> q = new LinkedList<>();
    String in;
    while(!(in = sc.next()).equals("quit"))
        if(in.equals("enqueue"))
            q.addLast(sc.next());
        else if(in.equals("dequeue"))
            if(q.size() == 0)
                System.out.println("queue is empty!");
            else
                System.out.println(q.removeFirst());
    System.out.println(q.size() + " elements left.");
}
```

---

10. Suppose you have two stacks of strings, one of them is empty and the other is not. You want to transfer the content from one to other without changing the order. Write a function which takes these stacks as arguments and performs this task. (You can assume that the first argument will always be the non-empty stack.)

Main method:

```
public static void main(String[] args) {
    Stack<String> s1 = new Stack<>();
    s1.push("a");
    s1.push("b");
```

```
    s1.push("c");
    Stack<String> s2 = new Stack<>();
    stackTransfer(s1, s2);
    System.out.println(s1);
    System.out.println(s2);
}
```

solution:

```
static void stackTransfer(Stack<String> s1, Stack<String> s2) {
    Stack<String> temp = new Stack<>();
    while(!s1.empty())
        temp.push(s1.pop());
    while(!temp.empty())
        s2.push(temp.pop());
}
```

---

11. Write a function which takes a string and checks if all of the parentheses (if any) in it are balanced.

Hint: Use stack

example:

- `isBalanced("(yeah)") = true`
- `isBalanced("(yeah)(i)am(ba(lan)ced)") = true`
- `isBalanced("((something)") = false`
- `isBalanced("no parenthesis") = true`
- `isBalanced("we(i)rd") = true`

Main method:

```
public static void main(String[] args) {
    System.out.println(isBalanced("(yeah)"));
    System.out.println(isBalanced("(yeah)(i)am(ba(lan)ced)"));
    System.out.println(isBalanced("((something)"));
    System.out.println(isBalanced("no parenthesis"));
    System.out.println(isBalanced("we(i)rd"));
}
```

solution:

```
static boolean isBalanced(String s) {
    // the kind of elements that you
    // push to the stack has no importance
    // in this question
    Stack<Object> st = new Stack<>();
    for(int i=0; i<s.length(); i++)
        if(s.charAt(i)=='(')
            st.push(null);
        else if(s.charAt(i) == ')')
            if(st.empty())
                return false;
            else
                st.pop();
    return st.empty();
}
```

---

12. Write a function which takes two stacks as parameters and swaps their contents without changing the order of the elements. You are only allowed to use primitive data types and one extra stack (don't use more stacks). Allowed operations on a stack are standard ops like push, pop, peek, size.

Main method:

```
public static void main(String[] args) {
    Stack<Integer> s1 = new Stack<>();
    s1.push(1);
    s1.push(2);
    s1.push(3);
    Stack<Integer> s2 = new Stack<>();
    s2.push(4);
    s2.push(5);
    s2.push(6);
    stackSwap(s1, s2);
    System.out.println(s1);
    System.out.println(s2);
}
```

solution:

```
static void transfer(Stack<Integer> s1, Stack<Integer> s2) {
    while(!s1.empty())
        s2.push(s1.pop());
}
static void stackSwap(Stack<Integer> s1, Stack<Integer> s2) {
    Stack<Integer> temp = new Stack<>();
    int s1Size = s1.size();
    transfer(s2, temp);
    transfer(s1, temp);
    for(int i=0; i<s1Size; i++)
        s2.push(temp.pop());
    transfer(temp, s1);
}
```

---

13. In your high-school, one of your friends has a rich dad and upon your friend's academic success, his dad decided to gift all the school with unlimited amount of baklava. Baklavas are distributed from school cafeteria one-by-one and initially 500 students are in the queue. Each time with 60% probability, the student in the front of the queue decides to eat one more piece and enters the queue back again from the back. Baklavas are served until there's no one left in the queue. A piece of baklava costs 7 liras.

Simulate this process and find out the average spending of rich guy for this festival.

Main method:

```
public static void main(String[] args) {
    System.out.println(baklavaMadness(500, 7, 0.6));
}
```

solution:

```
static int baklavaMadness(int numStudents, int baklavaCost, double p) {
    Queue<Object> q = new LinkedList<>();
    int baklavasEaten = 0;
    for(int i=0; i<numStudents; i++)
        q.add(null);
    while(!q.isEmpty()) {
        baklavasEaten++;
        if(Math.random() < p)
            q.add(q.remove());
        else
            q.remove();
    }
    return baklavasEaten * baklavaCost;
```

```
}
```

---

14. Write a function which takes a string representing an arithmetical expression written in postfix notation and calculates the result. All numbers are integers and only + and * operations are supported. You can assume that the sentence has correct syntax.

Hint: Use stack

example:

- `solve("2 4 +") = 6`
- `solve("2 1 + 1 1 + *") = (2+1) * (1+1) = 6`
- `solve("2 +")` – (no need to handle)

Main method:

```
public static void main(String[] args) {
    System.out.println(solve("2 4 +"));
    System.out.println(solve("2 1 + 1 1 + *"));
}
```

solution:

```
static int solve(String expr) {
    Stack<Integer> st = new Stack<>();
    for(String token: expr.split(" "))
        switch(token) {
            case "+":
                st.push(st.pop() + st.pop());
                break;
            case "*":
                st.push(st.pop() * st.pop());
                break;
            default:
                st.push(Integer.valueOf(token));
        }
    return st.pop();
}
```

Project

- You and your friends form a circle to play a game. The game starts with you picking either the friend on your right or the one on your left with equal probability and whispering the secret word to her ear. Game continues with that chosen friend taking your role and so on. The game ends when everybody in the circle knows the word. Simulate this game in Java. As a bonus try to come up with a formula relating the number of people in the circle and the average number of rounds the game takes.

- Recall the Round-Robin job scheduling algorithm. Represent each job's remaining execution time with a LinkedList. Write a program which simulates this scheduling algorithm, consider why using a LinkedList data structure is appropriate.

solution:

```java
public static void main(String[] args) {
    // Create a map of jobs with their remaining execution times
    Map<String, LinkedList<Integer>> jobs = new HashMap<>();
    // Job1 has 12 units of time remaining initially
    jobs.put("Job1", new LinkedList<>(Arrays.asList(12, 8, 4)));
    // Job2 has 9 units of time remaining initially
    jobs.put("Job2", new LinkedList<>(Arrays.asList(9, 6, 2)));
    // Job3 has 15 units of time remaining initially
    jobs.put("Job3", new LinkedList<>(Arrays.asList(15, 10, 6)));

    // Set the time slice for each job
    int timeSlice = 4;

    // Simulate the scheduling algorithm
    while (!jobs.isEmpty()) {
        // Iterate over a copy of the key set to avoid concurrent modification exceptions
        for (String job : new ArrayList<>(jobs.keySet())) {
            // Get the remaining execution time for the current job
            LinkedList<Integer> remainingTime = jobs.get(job);

            // If the job has no remaining execution time, remove it from the map
            if (remainingTime.isEmpty()) {
                jobs.remove(job);
            } else {
                // Otherwise, get the next unit of time to execute
                int time = remainingTime.removeFirst();

                // If the job can be completed in the current time slice, remove it from the map
                if (time <= timeSlice) {
                    System.out.println(job + " completed.");
                    jobs.remove(job);
                } else {
                    // Otherwise, add the remaining time back to the end of the list
                    remainingTime.addLast(time - timeSlice);
                    System.out.println(job + " has " + remainingTime.getFirst() + " units of time remaining.");
                }
            }
        }
    }
}
```

- You want to buy a card collection which consists of 30 different cards in total. Each card normally costs 3 liras. However shop assistant is too bored that day and proposed you another scheme to buy the cards. According to the alternative scheme, you would draw a card from a bag randomly (where probability of drawing any different card is always equal), and you are supposed to pay 1 lira for each draw whether or not you draw a new card or one you have already drawn

before. You are also allowed to switch between schemes at any time you want. What is the best strategy to buy the whole collection? Which strategy would you prefer if switching hadn't been allowed?

- Write a function which takes a HashMap<Integer, String> and removes duplicates in the value collection while preserving the mapping. That is for any integer n in the keyset, map(n) = s iff map'(n) = s where map' is the changed map.

Main method:

```
public static void main(String[] args) {
    // Create a sample HashMap
    HashMap<Integer, String> map = new HashMap<>();
    map.put(1, "apple");
    map.put(2, "banana");
    map.put(3, "cherry");
    map.put(4, "banana");
    map.put(5, "apple");

    // Print the original map
    System.out.println("Original map:");
    System.out.println(map);

    // Remove duplicate values from the map
    removeDuplicateValues(map);

    // Print the modified map
    System.out.println("Map with duplicates removed:");
    System.out.println(map);
}
```

solution:

```
public static void removeDuplicateValues(HashMap<Integer, String> map) {
    // Create a set to keep track of unique values
    HashSet<String> uniqueValues = new HashSet<>();

    // Create a list to keep track of keys with duplicate values
    ArrayList<Integer> keysToRemove = new ArrayList<>();

    // Iterate over the map entries
    for (Map.Entry<Integer, String> entry : map.entrySet()) {
        String value = entry.getValue();

        // If the value has already been seen, mark the key for removal
        if (uniqueValues.contains(value)) {
            keysToRemove.add(entry.getKey());
        } else {
            uniqueValues.add(value);
        }
    }

    // Remove the keys with duplicate values from the map
    for (Integer key : keysToRemove) {
        map.remove(key);
    }
}
```

- Write a function which takes a HashMap<Integer, Integer> and checks if it is a permutation mapping (in the same sense as the permutation function in math).

Main method:

```
public static void main(String[] args) {
    // Example 1: Permutation mapping
    HashMap<Integer, Integer> map1 = new HashMap<>();
    map1.put(1, 2);
    map1.put(2, 3);
    map1.put(3, 1);

    boolean isPermutation1 = isPermutation(map1);
    if (isPermutation1) {
        System.out.println("Example 1: The map is a permutation mapping.");
    } else {
        System.out.println("Example 1: The map is not a permutation mapping.");
    }

    // Example 2: Not a permutation mapping
    HashMap<Integer, Integer> map2 = new HashMap<>();
    map2.put(1, 2);
    map2.put(2, 3);
    map2.put(3, 3);

    boolean isPermutation2 = isPermutation(map2);
    if (isPermutation2) {
        System.out.println("Example 2: The map is a permutation mapping.");
    } else {
        System.out.println("Example 2: The map is not a permutation mapping.");
    }
}
```

Solution:

```
public static boolean isPermutation(HashMap<Integer, Integer> map) {
    // Create a set to keep track of seen values
    HashSet<Integer> seenValues = new HashSet<>();

    // Iterate over the map entries
    for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
        int key = entry.getKey();
        int value = entry.getValue();

        // Check if the key and value are equal
        if (key == value) {
            return false;
        }

        // Check if the value has already been seen
        if (seenValues.contains(value)) {
            return false;
        }

        seenValues.add(value);
    }

    // If we made it here, all values were unique and the keys and values were not equal
    return true;
}
```

- Write a function which takes two integers a and b, and a HashMap<String, Integer>, mapping each person's name to her

age, and returns another HashMap<String, Integer> which filters the input mapping and includes only those people with the age in between a and b. You can assume that a<=b.

Main method:

```
public static void main(String[] args) {
    HashMap<String, Integer> people = new HashMap<>();
    people.put("Alice", 25);
    people.put("Bob", 30);
    people.put("Charlie", 20);
    people.put("David", 35);

    HashMap<String, Integer> filtered = filterByAge(25, 30, people);
    System.out.println(filtered);  // prints "{Alice=25, Bob=30}"
}
```

Solution:

```
public static HashMap<String, Integer> filterByAge(int a, int b, HashMap<String, Integer> people) {
    // Create a new HashMap to store the filtered results
    HashMap<String, Integer> filtered = new HashMap<>();

    // Loop over the input mapping and check the age of each person
    for (String name : people.keySet()) {
        int age = people.get(name);

        // If the age is in the range [a, b], add the person to the filtered mapping
        if (age >= a && age <= b) {
            filtered.put(name, age);
        }
    }

    // Return the filtered mapping
    return filtered;
}
```

- Write a function which takes a Stack as parameter, returns the second element (from top) without changing the stack at the end.

Main method:

```
public static void main(String[] args) {
    // Create a stack of integers
    Stack<Integer> stack = new Stack<>();
    stack.push(1);
    stack.push(2);
    stack.push(3);
    stack.push(4);

    // Get the second element of the stack
    int second = getSecondElement(stack);
    System.out.println("The second element of the stack is: " + second);

    // Verify that the original stack is unchanged
    System.out.println("The original stack is: " + stack);
}
```

Solution:

```
public static int getSecondElement(Stack<Integer> stack) {
    int top = stack.pop();  // remove top element
```

```
    int second = stack.peek();  // get second element without removing it
    stack.push(top);  // push back the original top element
    return second;
}
```

- Write a function which takes two string stacks of equal size and interleaves them into a single stack and returns it. > example

interleave([1,2,3], [a,b,c]) returns [1,a,2,b,3,c]

Main method:

```
public static void main(String[] args) {
    Stack<String> s1 = new Stack<>();
    s1.push("a");
    s1.push("b");
    s1.push("c");
    Stack<String> s2 = new Stack<>();
    s2.push("1");
    s2.push("2");
    s2.push("3");
    Stack<String> result = interleave(s1, s2);
    System.out.println(result); // prints "[c, 3, b, 2, a, 1]"
}
```

Solution:

```
public static Stack<String> interleave(Stack<String> s1, Stack<String> s2) {
    Stack<String> result = new Stack<>();
    while (!s1.isEmpty() && !s2.isEmpty()) {
        result.push(s1.pop());
        result.push(s2.pop());
    }
    // If one of the stacks is longer, add the remaining elements to the result stack
    while (!s1.isEmpty()) {
        result.push(s1.pop());
    }
    while (!s2.isEmpty()) {
        result.push(s2.pop());
    }
    return result;
}
```

- Write a function which takes two stacks of sorted integers and dumps their content in a new stack in such a way that the resulting stack is again sorted but in reverse direction.

Main method:

```
public static void main(String[] args) {
    // Create two sorted stacks
    Stack<Integer> stack1 = new Stack<>();
    stack1.push(10);
    stack1.push(7);
    stack1.push(5);
    stack1.push(3);
    stack1.push(1);

    Stack<Integer> stack2 = new Stack<>();
    stack2.push(9);
```

```
    stack2.push(6);
    stack2.push(4);
    stack2.push(2);

    // Merge the stacks in reverse order
    Stack<Integer> resultStack = reverseSortedMerge(stack1, stack2);

    // Print the merged stack
    System.out.println(resultStack);
}
```

Solution:

```
public static Stack<Integer> reverseSortedMerge(Stack<Integer> stack1, Stack<Integer> stack2) {
    Stack<Integer> resultStack = new Stack<>();

    // Merge the two stacks in reverse order
    while (!stack1.empty() && !stack2.empty()) {
        if (stack1.peek() < stack2.peek()) {
            resultStack.push(stack1.pop());
        } else {
            resultStack.push(stack2.pop());
        }
    }

    // Dump the remaining elements of stack1 into resultStack
    while (!stack1.empty()) {
        resultStack.push(stack1.pop());
    }

    // Dump the remaining elements of stack2 into resultStack
    while (!stack2.empty()) {
        resultStack.push(stack2.pop());
    }

    return resultStack;
}
```

- Write a function which takes a stack and returns the bottom-most element without changing the stack in the end.

Main method:

```
public static void main(String[] args) {
    // Create a stack of integers
    Stack<Integer> stack = new Stack<>();
    stack.push(1);
    stack.push(2);
    stack.push(3);
    stack.push(4);
    stack.push(5);

    // Call the getBottom function to retrieve the bottom-most element
    int bottom = getBottom(stack);

    // Print the bottom-most element
    System.out.println("Bottom-most element: " + bottom);
}
```

Solution:

```java
public static int getBottom(Stack<Integer> stack) {
    // Check if the stack is empty
    if (stack.isEmpty()) {
        throw new IllegalArgumentException("Stack is empty");
    }

    // Pop all elements from the stack and store them in a temporary stack
    Stack<Integer> tempStack = new Stack<>();
    while (!stack.isEmpty()) {
        tempStack.push(stack.pop());
    }

    // Get the bottom-most element
    int bottom = tempStack.peek();

    // Push all elements back onto the original stack
    while (!tempStack.isEmpty()) {
        stack.push(tempStack.pop());
    }

    // Return the bottom-most element
    return bottom;
}
```

Extra

- You have a bag which contains n black, n red balls. As your choice of n gets bigger, doing the following experiment many times and taking the average will yield a certain value, what is it?

Experiment: Randomly draw balls from the bag one-by-one until one of the colors is completely depleted. Then count the remaining balls in the bag.

Main method:

```
public static void main(String[] args) {
    Random r = new Random();
    LinkedList<String> balls = new LinkedList<>();
    for(int i = 0; i < 100; i++) {
        if (r.nextBoolean())
            balls.add("red");
        else
            balls.add("black");
    }
    System.out.println(drawBalls(balls));
}
```

solution:

```
static int drawBalls(LinkedList<String> balls) {
    Random r = new Random();
    int redBalls = balls.size() / 2;
    int blackBalls = balls.size() / 2;

    while(redBalls > 0 && blackBalls > 0) {
        int randBall = r.nextInt(balls.size());
        if(balls.get(randBall) == "red")
            redBalls--;
        else
            blackBalls--;
        balls.remove(randBall);
    }
    return redBalls + blackBalls;
}
```

- Consider a group of kids forming a queue to shoot a basket in the schoolyard. All kids begin with 0 point, and each successful basket accounts +1 point. The kid in the front of the queue shoots (potentially more than once) until she fails to score a basket, after which she enters the queue from the back again. Each kid has 30% probability of shooting a successful basket each time. The game continues until one of the kids reaches 10 points. Write a program to simulate this game. Experiment with different values for the total number of kids and try to see how it affects the average number of turns a game takes from beginning to end.

solution:

```
// Probability that a kid will make a basket
private static final double BASKET_PROBABILITY = 0.3;

// Number of points needed to win the game
private static final int WINNING_SCORE = 10;

// Random number generator for shooting attempts
private static final Random RANDOM = new Random();

public static void main(String[] args) {
    // Number of kids in the queue
    int numKids = 10;
```

```
    // Number of times to run the simulation
    int numSimulations = 10000;

    // Total number of turns across all simulations
    int totalTurns = 0;

    // Run the simulation multiple times and track the total number of turns
    for (int i = 0; i < numSimulations; i++) {
        int turns = simulateGame(numKids);
        totalTurns += turns;
    }

    // Calculate the average number of turns across all simulations
    double averageTurns = (double) totalTurns / numSimulations;
    System.out.println("Average number of turns: " + averageTurns);
}

/**
 * Simulates a game with a queue of kids, each shooting baskets until one kid
 * reaches the winning score.
 *
 * @param numKids the number of kids in the queue
 * @return the number of turns it took for the game to end
 */
private static int simulateGame(int numKids) {
    // Create a queue of kids, each with a score of 0
    Queue<Integer> queue = new LinkedList<>();
    for (int i = 0; i < numKids; i++) {
        queue.add(0);
    }

    // Track the number of turns in the game
    int numTurns = 0;

    // Continue playing until a kid reaches the winning score
    while (true) {
        // The kid in front of the queue shoots until they miss
        int score = 0;
        while (true) {
            // Attempt to make a basket
            if (RANDOM.nextDouble() < BASKET_PROBABILITY) {
                score++;
            } else {
                break;
            }
        }

        // Update the kid's score
        int currentScore = queue.remove() + score;
        queue.add(currentScore);

        // Check if any kid has reached the winning score
        for (int kidScore : queue) {
            if (kidScore >= WINNING_SCORE) {
                return numTurns;
            }
        }
```

```
        // Move the kid who just shot to the back of the queue
        queue.add(queue.remove());

        // Increment the turn counter
        numTurns++;
    }
}
```

- Imitate a queue using two stacks.