1. (a) Take an initially empty hash table with five slots, with hash function $h(x) \equiv_5 x$, and with collisions resolved by chaining. Draw a sketch of what happens when inserting the following sequence of keys into it: 35, 2, 18, 6, 3, 10, 8, 5. Hints:

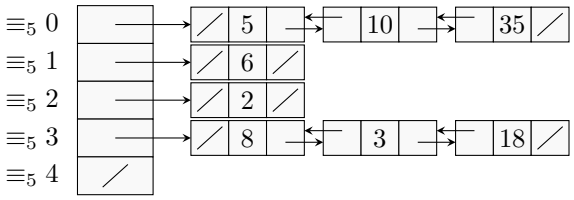
   You are not requested to draw the intermediate stages as separate figures, nor to show all the fields of each entry in detail.

   Insertion into chained hash tables is simple. Hash the key, insert the key, value into the address which the key hashed to. If the address is occupied; then depending on implementation you can either traverse to the end of the linked list and then insert the key-value at the end of the linked list or insert the key-value into the head of the linked list.

   The expected cost of traversing to the end of the linked list is $\Theta\left(\frac{n}{\alpha}\right)$ where n is the number of nodes in the hash table and α is the loading factor. I would therefore suggest that the standard implementation should therefore insert the key-value pair into the head of the list in chaining.

   Although there are cases where traversing to the end of the linked list makes more sense. For example if the elements put into the hash table first are more common: take an example from machine learning when we are storing the number of occurrences of words.

   In the implementation below I insert at the head of the chain. If we were to insert at the end of the linked list then the order of each linked list would be reversed.



   (b) Insert the same items with the following three changes: the hash table now has ten slots, the hash function is $h(x) = x \bmod 10$, and collisions are resolved by linear probing.

$\equiv_{10} 0$	10
$\equiv_{10} 1$	⁄
$\equiv_{10} 2$	2
$\equiv_{10} 3$	3
$\equiv_{10} 4$	⁄
$\equiv_{10} 5$	35
$\equiv_{10} 6$	6
$\equiv_{10} 7$	5
$\equiv_{10} 8$	18
$\equiv_{10} 9$	8

   (c) Imagine a hash table implementation where collisions are resolved by chaining but all the data stays within the slots of the original table. All entries not containing key–value pairs are marked with a Boolean flag and linked together into a free list.

   Give clear explanations of how to implement the set(key, value) method in expected constant time, highlighting notable points and using high-level pseudocode where appropriate. Make use of doubly-linked lists if necessary.

   Assume the hash table has 5 slots, is initially empty and uses the hash function $h(x) \equiv_5 x$. Draw five diagrams of the hash table representing the initially empty

state and then the table after the insertion of each of the following key–value pairs: $(2, A), (2, C), (12, T), (5, Z)$. In the final diagram, draw all the fields and pointers of all the entries.

Every node will have a pointer to the previous element in the linked list, a pointer to the next element in a linked list, a boolean indicating whether it contains a key-value pair or not and if it contains a key-value pair then it will have elements for the key and value.

Firstly I will define "relocating" a key-value pair. We take a node n which contains the key-value pair we wish to relocate. We first look at the linked list of free nodes and pop the head of this. We change it's key-value pair to the key-value pair we wish to relocate, copy the old nodes predecessor and successor pointer across into the new node and change the predecessors successor pointer and successors predecessor pointer to point to the new node. We then place the old node into the free list. This operation takes $\Theta(1)$ time. Note we first copy the node and then change pointers to cut out the old node. This ensures the chain is never severed and if the computer crashes mid-execution we run no risk of losing a chain of values.

To implement set(key, value) we must first hash the key and traverse to the address in the hash table the key hashed to. There are then 5 cases.

(1) There is no key, value in the position. In this case we change the key, value pair in the node at that position, remove it from the linked list of free nodes and set it's predecessor and successor pointers to itself (in my implementation all linked lists are circularly linked for simplicity – it reduces the number of edge cases you need to handle at no cost).

(2) There a node at that address but it was relocated. This means that no key has hashed to this address and so we are guaranteed there is no node with the same key as us. So we relocate the key-value which is at the address, then revert to case (1). Note that we relocate first so the chain in which the other address is stored is never severed.

(3) There is a node at that address and it was not relocated. In this case keys have hashed to the same value as our key. So there is a possibility that a node with the same key as us has hashed there and is in the linked list. Hence, in order to ensure we do not have duplicate keys, we must search through the entire linked list to ensure no keys match our key: if a key matches then we should replace the value of that element and terminate. So we pass through the circularly linked list until either we find a node with a key that matches ours – and replace its value. Or we reach the original location again and conclude that our key is not in the hash table. After that we relocate our key, value making it the second element in the linked list.

(4) One of the previous cases tries to relocate a node but the free list is empty. In this case we must make a new, larger array and hash function, then insert every node in the old hash table into the new one then delete the old array. This is has a linear time complexity. However this case can only happen once every n set operations. So set is amortized constant.

Here is a full implementation of this hash table in Java. I have commented the code as required.

```
import java.util.ArrayList;

public class ClosedChainHashTable<T> {

/*
initialising the required variables for the hash table.
table stores the elements themselves (I had to use an ArrayList
since arrays do not work with generics).
freelist is the head of the doubly linked circular list holding
all the free elements. It needs to be doubly linked and circular
so that we can extract from it in constant time (when inserting
directly into an unused location via a hash).
*/
    private ArrayList<Element> table;
    private Element freelist;
    private int contains = 0;
    private int size;

    private class Element {

        /*
        This initialises an element into an empty element. We require
        every element to be either free or filled. free is technically
        unneccessary since we could check whether it held a key, value —
        however having a boolean to indicate this is clearer and cleaner.
        key holds the key of the node. If the node is free then key=0
        (however this is never checked)
        value holds the payload of the element. It has arbitrary type
        since this uses generics. If we delete the key then value is set
        to null.
        index holds the nodes own index (a pointer to itself).
        next holds the index of the nodes successor
        previous holds the index of the nodes predecessor
        */
        private boolean free = true;
        private int key;
        private T value;
        private final int index;
        private int previous;
        private int next;
        private boolean relocated;

        private Element(int index, int previous, int next){
            assert index >= 0;
            assert previous >= 0;
            assert next >= 0;
            this.next = next;
            this.previous = previous;
            this.index = index;
        }

        private void fill(int key, T value, int previous, int next,
                boolean relocated){
            /*
```

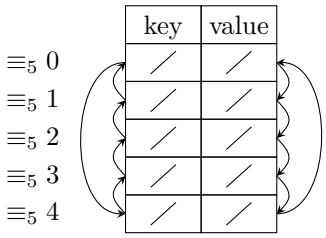
```
            This method changes the key, value, successor
            and predecessor of an Element.
            */
            if (freelist.index == index){
                freelist = table.get(freelist.next);
            }
            assert previous >= 0;
            assert next >= 0;
            table.get(this.previous).next = this.next;
            table.get(this.next).previous = this.previous;
            this.key = key;
            this.value = value;
            this.previous = previous;
            this.next = next;
            table.get(this.next).previous = index;
            table.get(this.previous).next = index;
            this.free = false;
            this.relocated = relocated;
        }

        private void clear(){
            /*
            This empties the node, removes it from it's
            linked list and adds it to the freelist.
            */
            value = null;
            key = 0;
            this.next = freelist.index;
            this.previous = freelist.previous;
            table.get(previous).next = index;
            table.get(next).previous = index;
            relocated = false;
            free = true;
            freelist = this;
        }
    }

    public ClosedChainHashTable(int table_size){
        assert table_size >= 1;
        this.size = table_size;
        table = new ArrayList<>(size);
        Element element;
        for (int i = 0; i < size; i++) {
            element = new Element(i, (i + size - 1) % size,
                        (i + 1) % size);
            table.add(element);
        }
        freelist = table.get(0);
    }

    public void add(int key, T value) {
        /*
        First I parse the linked list for our key to avoid any
        duplicate keys. From this point we can consider that the
        method has ended or the key is not in the table.
        */
```

```
        int hash_value = hash_function(key);
        Element current = table.get(hash_value);
        do {
                if (!current.free && current.key == key) {
                        current.value = value;
                        return;
                }
                current = table.get(current.next);
        }
        while (current.index != hash_value);

        /*
        If the hash table is full then we must resize the whole
        table and re-insert every element. This is an expensive
        linear time operation however it can only occur once
        every n calls to add. So the amortized cost is constant.
        */
        if (contains >= size) {
            size *= 2;
            ArrayList<Element> oldtable = table;
            table = new ArrayList<>(size);
            for (int i = 0; i < size; i++) {
                table.set(i, new Element(i, (i + size) % size,
                                (i + 1) % size));
            }
            for (Element element: oldtable) {
                add(element.key, element.value);
            }
        }
        /*
        Now I check the first element of the linked list.
        If it is empty then I can fill it without issue.

        If it was relocated, then I should relocate it again
        and insert my key, value into that position.

        If it was hashed to that address, then I should make
        my key, value it's successor.
        */
        int position = hash_function(key);
        Element node = table.get(position);
        if (node.free) {
            node.fill(key, value, position, position, false);
            contains++;
        }
        else if (node.relocated) {
            freelist.fill(node.key, node.value, node.previous, node.next, true);
            node.clear();
            node.fill(key, value, position, position, false);
            contains++;
        }
        else {
            freelist.fill(key, value, position, node.next, true);
            contains++;
        }
    }
```

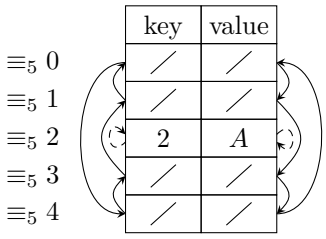
```
public T get(int key) {
    /*
    We hash the value and search along the linked list
    until we return to the address the key hashed to.
    */
    int hashvalue = hashfunction(key);
    int position = hashvalue;
    do{
        if (table.get(position).key == key){
            return table.get(position).value;
        }
        else{
            position = table.get(position).next;
        }
    }
    while (hashvalue != position);
    return null;
}

private int hashfunction(int k){
    return k % size;
}


public void delete(int key){
    /*
    If the value the key hashes to is the element we
    wish to delete, then we must replace it with another
    element in its linked list (if any).
    If not then we should parse through the linked list
    until either we return to the address the key hashed
    to or find the key. If we do find the key, we should
    remove the node from its linked list and de-allocate
    its key, value.
    */
    int hash_value = hash_function(key);
    Element current = table.get(hash_value);
    if (current.free) {
        return;
    }
    if (current.key == key) {
        Element successor = table.get(current.next);
        current.value = successor.value;
        current.key = successor.key;
        current.next = successor.next;
        table.get(successor.next).previous = current.index;
        successor.clear();
        contains--;
    }
    else {
        int position = current.next;
        while (position != hash_value) {
            current = table.get(position);
            if (current.key == key) {
                current.value = null;
```

```
                    current.key = 0;
                    table.get(current.previous).next = current.next;
                    table.get(current.next).previous = current.previous;
                    current.next = freelist.index;
                    current.previous = freelist.previous;
                    table.get(freelist.previous).next = current.index;
                    freelist.previous = current.index;
                    freelist = current;
                    contains--;
                } else {
                    position = table.get(position).next;
                }
            }
        }
    }
}
```

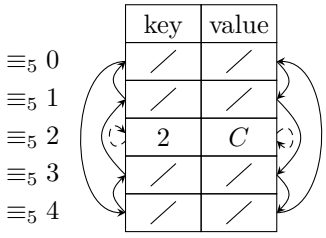
In the diagram below, the pointers on the right are successor pointers, those on the left are predecessor pointers and a strike through the key or value means that no value is stored there.



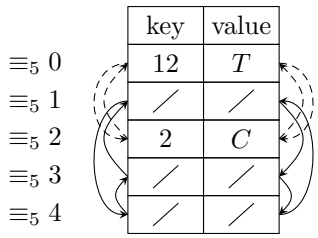
Firstly we insert $2, A$. The key hashes to 2. The node at position 2 is empty – and so we insert $2, A$ at position 2 and change the relevant pointers.



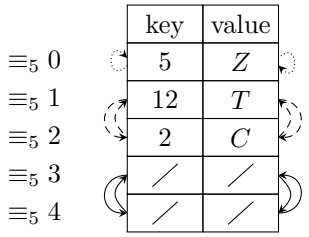
Next, we insert $2, C$. However, when we hash 2, we reach position 2 – which is occupied by $2, A$. Note that they share a key and hash tables do not support duplicate keys. So we must replace A with C. This does not involve changing any pointers.



Now we insert $12, T$. This also maps to 2. When we look, $2, C$ and $12, T$ do not have the same key. So we insert T into the linked list following $2, C$. The first element in the list of free elements is at position 0. So we insert T into position 0 and change the relevant pointers.

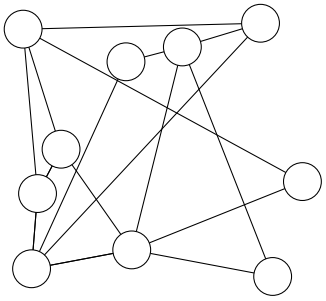


Now we insert $5, Z$. This hashes to 0. Position 0 is occupied by $12, T$. This was not hashed to 0 – so we relocate $12, T$. The head of the free list is at position 1. So $12, T$ is relocated to 1. Now that the slot at 0 is free, we can insert $5, Z$ there.



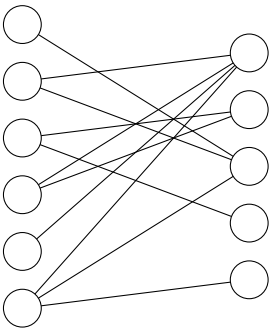
1. Explain what is meant by the terms directed graph, undirected graph and bipartite graph.

   A directed graph is a graph where edges have directions.



   An undirected graph is a graph where edges do not have directions.

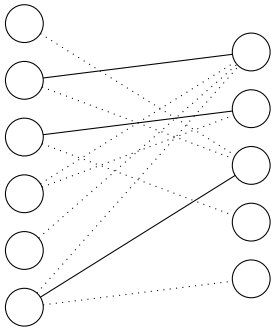
   A bipartite graph is graph which can be divided into two disjoint groups such that no node in the right group can reach any other node in the right group and no node in the left group can reach any node in the left group.



2. Given a bipartite graph, what is meant by a matching, and what is an augmenting path with respect to a matching?

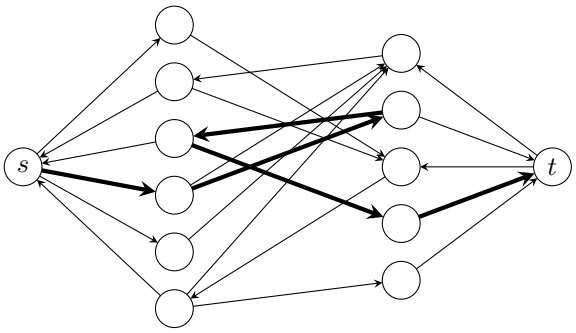
A matching is a subset of edges in a bipartite graph which connects every node in the left set to zero or one nodes in the right set and every node in the right set to zero or one nodes in the left set.

In the below diagrams, arrows indicate direction. Dotted lines have weight 0 and full lines have weight 1.



An augmenting path to a matching m is a path in the residual graph to the which reaches the sink.

Let the bold arrows be the augmenting path in the residual graph formed by the matching above the last question.



3. Prove that if no augmenting path exists for a given matching then that matching is maximum.

   Assume there is no augmenting path. This means that the sink is not reachable from the source in the residual graph.

   Consider the set of all nodes which are reachable from the source as a cut. The source and the sink are in different sets. Assume we wish to traverse from the source to the sink, the path we must take must cross this cut at some point. So

   The capacity of this cut is equal to the flow from all nodes in the cut to all nodes which are not in the cut. This is equal to the current flow of the graph. We know that the maximal flow is less than or equal to the capacity of any cut (where the source and sink are on different sides of the cut). So the maximal possible flow must be less than or equal to the current flow. So the current flow is equal to the maximal flow.

   So the matching is maximum as required.

4. Outline an algorithm based on this property to find a maximal matching, and estimate its cost in terms of the number of vertices n and edges e of the given bipartite graph.

   Relate every node in the left set to a node s with an edge with capacity 1. Then relate every node in the right set to a node t with an edge with capacity 1. Set the capacity of all edges in the original graph to 1. Run Ford-Fulkerson on the new graph with source s and sink t.

When run on a graph with integral capacities, Ford-Fulkerson is guaranteed to result in every edge having an integral flow. Since each flow corresponds with two further nodes being contained in the matching, we know that the graph with the highest flow is isomorphic to the largest matching and Ford Fulkerson is proven to return the graph with the highest flow and this means the resultant flow graph can be converted into the largest matching.

If an edge between a node in the left graph and the right graph has flow 1 then the two nodes are related in the matching.

The worst-case cost of Ford-Fulkerson is $\Theta(f^*E)$ where f^* is the maximal flow. This is because we know that for each bfs, Ford-Fulkerson will augment the flow by an integral amount (on graphs where the flow is integral). This means that there are at worst $f^* + 1$ bfs's. The complexity of a bfs is $\Theta(V + E)$. However, when we run Ford-Fulkerson, we know that the graph is connected meaning that, $\Theta(V) \in O(E)$ and so the complexity is $\Theta(f^*E)$.

Note that in a matching with V vertices, the highest possible flow rate ($f^*_{\max}$) is $\frac{V}{2}$ and occurs when every vertex is involved in the matching. So the worst case of ford-fulkerson on a matching is $\Theta(VE)$.

In this graph there are n nodes and e edges: so the worst-case complexity of this algorithm is $\Theta(ne)$.