يوم الامتحان: الثلاثاء

المستوي الثالث (حاسب)

تاريخ الامتحان: ١٧ / ١ / ٢٠١٧ م

المادة : موضوعات مختارة في علوم الحاسب ١ (٣٥٥ رس)

الممتحن: د/ مصعب عبد الحميد محمد حسان

مدرس بقسم الرياضيات بكلية العلوم

الاسئلة و نموذج الإجابة

ورقة كاملة

Benha University
Faculty of Science
Dept. of Mathematics

Time: Two Hours
First Semester 2016-2017
Date : 17/1/2017

Selected Topics in Computer Science (1) (MC355) for Third Level Students (Computer Science)

Answer the following questions:

Question 1. (16 marks)

A- Define AVL tree, hashing, and directed graph. (4 marks)

B- Discuss graph invariants in details. (8 marks)

C- Write a function to apply the right rotation round node p. (4 marks)

Question 2. (17 marks)

A- Draw the binary search tree by inserting the following sequence into an initially empty tree:

26 15 29 14 20

Test if this tree is AVL tree or not? Suppose we insert a new node of data 17, test that the tree is AVL tree or not and if the tree is not AVL tree show how can we rebalance it? (5 marks)

B- Compare between adjacency matrix representation and adjacency list representation of graphs. (4 marks)

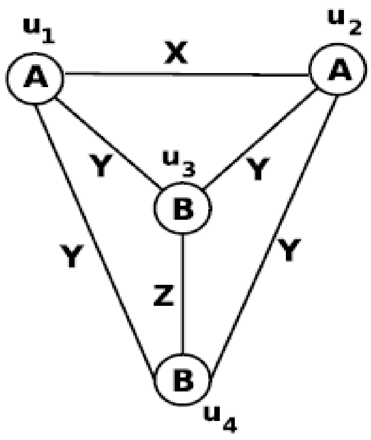
C- Discuss collision resolution strategies. (8 marks)

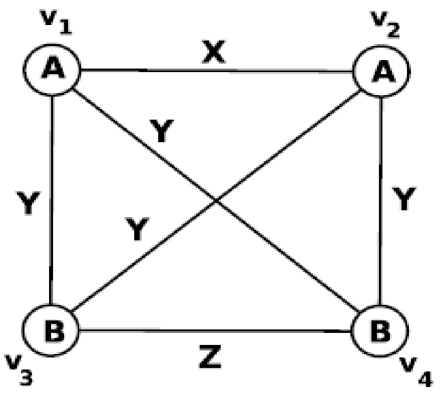
Question 3. (15 marks)

A- Discuss the applications of hashing. (3 marks)

B- Using adjacency matrix representation, write a function to test if the given graph is complete or not. (4 marks)

C- Using Ullman algorithm, test that the following two graphs are isomorphic or not. (8 marks)





Best Wishes
Dr. Mosab Abd El-Hameed

Model Answer

Selected Topics in Computer Science (1) (MC355) for Third Level Students (Computer Science)

Answer of Question 1.

A- AVL tree: A binary search tree in which the balance factor of each node is 0, 1, or -1, where the balance factor of a node x is the height of the left subtree of x minus the height of x's right subtree. (Recall that the height of a tree is the number of levels in it.)

Hashing: allows us to update and retrieve any entry in constant time O(1).

directed graph: A graph G = (V, E) is a non-linear data structure consists of a set of vertices V together with a set E of vertex pairs or edges. Each edge in the set E is a pair (x, y), where x and y belongs to V. If the edge pair is ordered, the edge is called directed and thus the graph is directed graph.

B- A graph invariant is a function T such that if applied to two isomorphic graphs H and G, then T(H) = T(G). In other words, if T(H) ≠ T(G) then H is not isomorphic to G.

The following are a graph invariants for graph isomorphism problem

1- Number of nodes of a given graph

2- Number of edges of a given graph

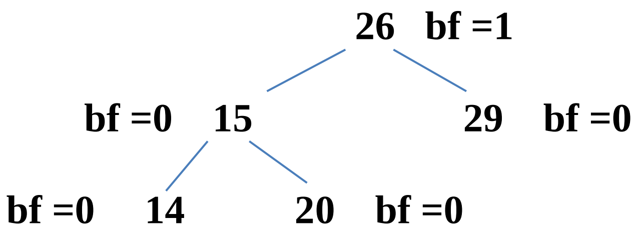
3- Number of cycles of a given graph (Number of paths that the first vertex is identical to the last vertex)

4- Degree sequence of a given graph (list the degree of each vertex in this graph), and so on

C-
```
node* BST::rotateright(node* p) // the right rotation round p
 {
   node* q = p->left;
   p->left = q->right;
   q->right = p;
   return q;
 }
```

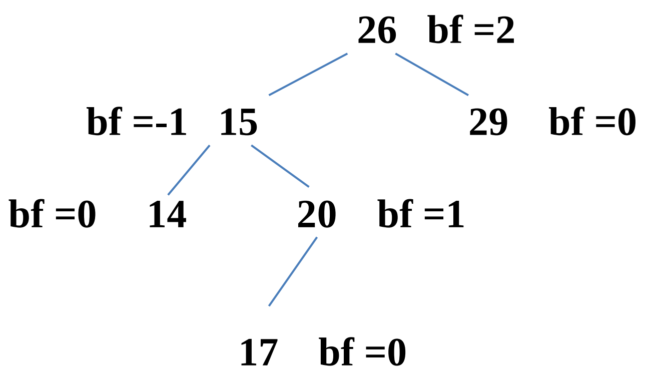
Answer of Question 2.

A-



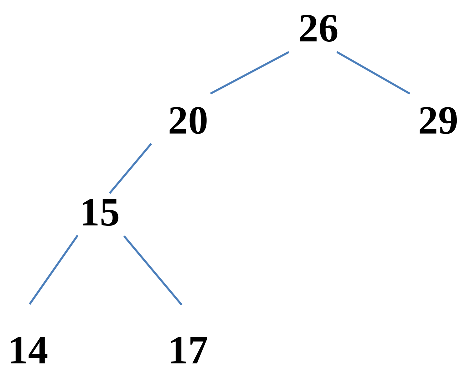
The Above tree is AVL tree.

After inserting a new node of data 17, the tree is not AVL tree

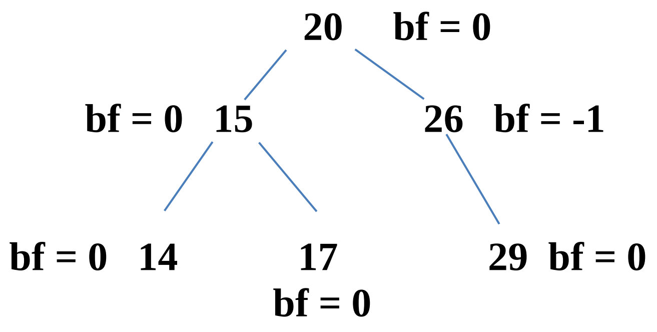


we can we rebalance it using left-right rotation.

First we perform a left rotation of the nodes in the left subtree of the nearest ancestor with balance factor 2



Now we apply a right rotation to the tree



B-

Comparison	Winner
Faster to test if (x, y) is in graph?	adjacency matrices
Faster to find the degree of a vertex?	adjacency lists
Less memory on small graphs?	adjacency lists
Less memory on big graphs?	adjacency matrices
Edge insertion or deletion?	adjacency matrices
Faster to traverse the graph?	adjacency lists
Better for most problems?	adjacency lists

Table : Relative advantages of adjacency lists and matrices.

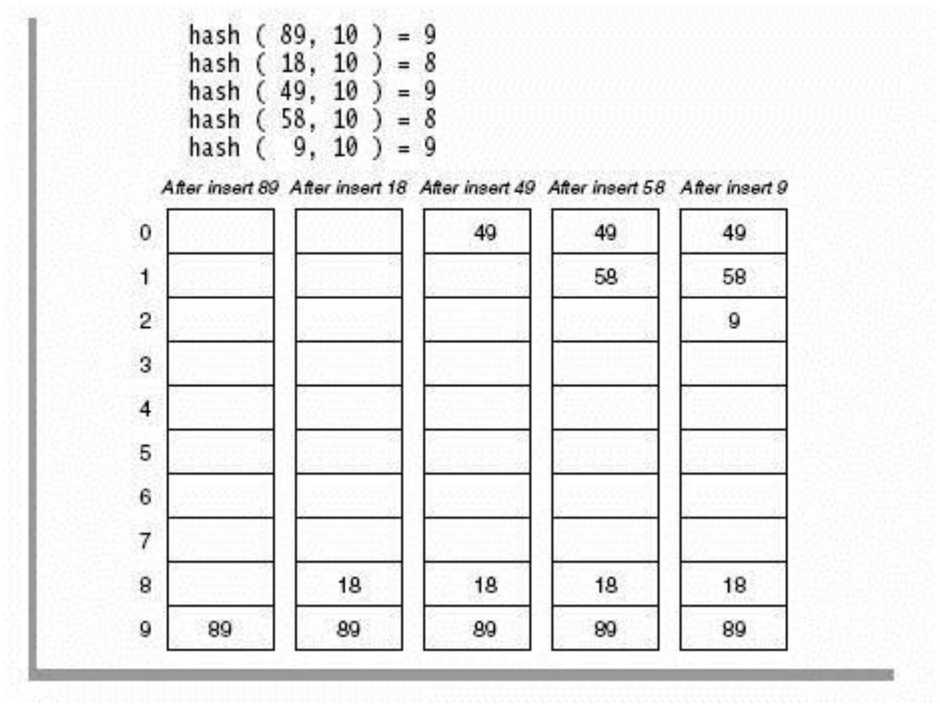
C-

Collision Resolution

Here we discuss two strategies of dealing with collisions, linear probing and separate chaining.

Linear Probing

Suppose that a key hashes into a position that is already occupied. The simplest strategy is to look for the next available position to place the item. Suppose we have a set of hash codes consisting of {89, 18, 49, 58, 9} and we need to place them into a table of size 10. The following table demonstrates this process.

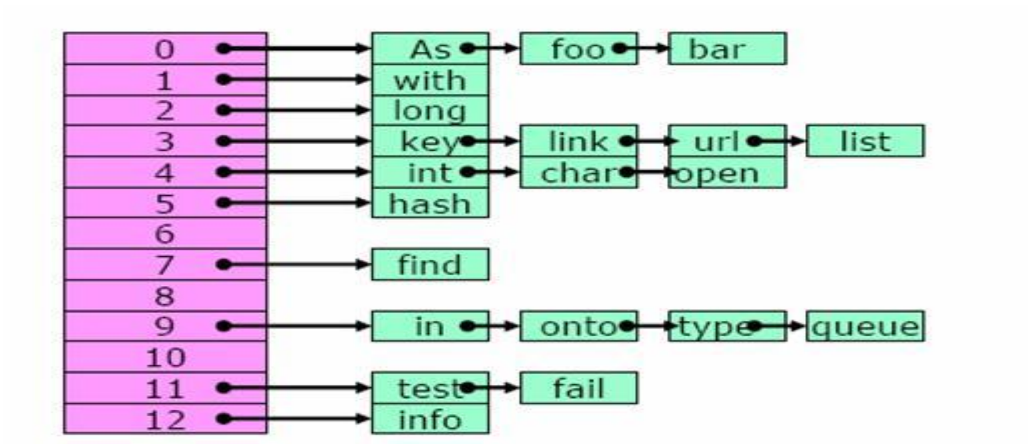


	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

The first collision occurs when 49 hashes to the same location with index 9. Since 89 occupies the A[9], we need to place 49 to the next available position. Considering the array as circular, the next available position is 0. That is (9+1) mod 10. So we place 49 in A[0]. Several more collisions occur in this simple example and in each case we keep looking to find the next available location in the array to place the element. Now if we need to find the element, say for example, 49, we first compute the hash code (9), and look in A[9]. Since we do not find it there, we look in A[(9+1) % 10] = A[0], we find it there and we are done. So what if we are looking for 79? First we compute hashcode of 79 = 9. We probe in A[9], A[(9+1)%10]=A[0], A[(9+2)%10]=A[1], A[(9+3)%10]=A[2], A[(9+4)%10]=A[3] etc. Since A[3] = null, we do know that 79 could not exists in the set.

Separate Chaining

The last strategy we discuss is the idea of separate chaining. The idea here is to resolve a collision by creating a linked list of elements as shown below.



In the picture above the objects, "As", "foo", and "bar" all hash to the same location in the table, that is A[0]. So we create a list of all the elements that hash into that location. Similarly, all other lists indicate keys that were hashed into the same location. Obviously a good hash function is needed so that keys can be evenly distributed. Because any uneven distribution of keys will neutralize any advantage gained by the concept of hashing. Also we must note that separate chaining requires dynamic memory management (using pointers) that may not be available in some programming languages. Also manipulating a list using pointers is generally more complicated than using a simple array.

Answer of Question 3.

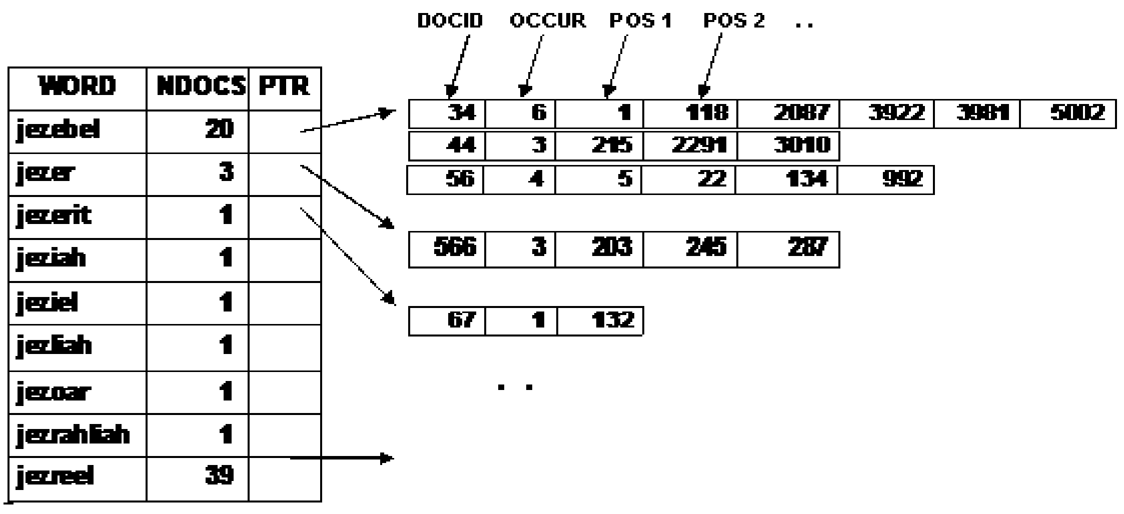
A- Applications of Hashing

In the internet age, hashing has actually become the data structure of convenience for many applications. Most modern object oriented

programming languages like Java and C# comes with built in hash functions that can be used to compute the hash value of any object. The advantage of hashing is that we can hash just about anything, strings, tables, complex data structures etc. This allows us to store and retrieve data in O(1) time.

Web Index

One possible application of hashing is saving a web index as a hash table. An inverted web index is a mapping between "keywords" and "URL's" as shown below.



The above data structure is built using a web crawl where a process collect pages and key words from each page and organize them in a data structure as above. So if we need to find all documents that are connected to a particular keyword, we simply hash into the keyword and find the corresponding list.

B-
```
Bool Test_Complete(int V, int E){
            int m = V * (V - 1) / 2 ;
            if(m == E)
              return true;
            else
              return false;
    }
```

C- Ullman algorithm is the earliest and highly-cited approach to the graph isomorphism problem. Given two graphs G1 and G2. To check if G1 is subgraph of G2, Ullman's basic approach is to enumerate all possible mappings of vertices in V_{G1} to those in V_{G2} using a depth-first

tree-search algorithm. In order to cope with graph isomorphism problem efficiently, Ullman proposed a refinement procedure to prune the search space. It is based on the following three conditions:

1. ***Label and degree condition.***

A vertex u $\in V_{G1}$ can be mapped to v $\in V_{G2}$ under bijective mapping f, i.e v = f(u), if

(i) $L_{G1}(u) = L_{G2}(v)$, and

(ii) $\deg_{G1}(u) = \deg_{G2}(v)$.

In our example, we can find the map f(u1) = v1, f(u2) = v2, f(u3) = v3, and f(u4) = v4 where $L_{G1}(u1) = L_{G2}(v1)$ =A and $\deg_{G1}(u1) = \deg_{G2}(v1) = 3$ and so on

2. ***One-to-One mapping of vertices condition.***

Once vertex u $\in V_{G1}$ is mapped to v $\in V_{G2}$, we cannot map any other vertex in V_{G1} to the vertex v $\in V_{G2}$.

3. ***Neighbor condition.***

By this condition Ullman algorithm examines the feasibility of mapping u $\in V_{G1}$ to v $\in V_{G2}$ by considering the preservation of structural connectivity. If there exist edges connecting u with previously explored vertices of G1 but there are no counterpart edges in G2, the mapping test simply fails. In our example, there is edge between u1 and u2 in G1 then there is counterpart edge between f(u1) = v1 and f(u2) = v2 in G2 and so on.