Problem2:

Disadvantages of Binary search Trees over hashtable:

1. For Search/Insert and Delete operations time complexity for Binary Trees is O(nlogn) in average case. In worst case i.e. in case of skewed or unbalanced trees it's O(n). On the other hand, Hashtable has time complexity is O(1) for performing same operations.

2. This is due to the fact that Hashtable uses hash function to calculate key to navigate to particular location and then perform above mentioned operations hence size of the data structure does not matter as far as searching, inserting and deletion are concerned.

3. When we know the size of data in advance and there is no requirement to sort this data then it's a good idea to use hashtable as its space requirement is not as high as BST.

Advantages of Binary search Trees over hashtable:

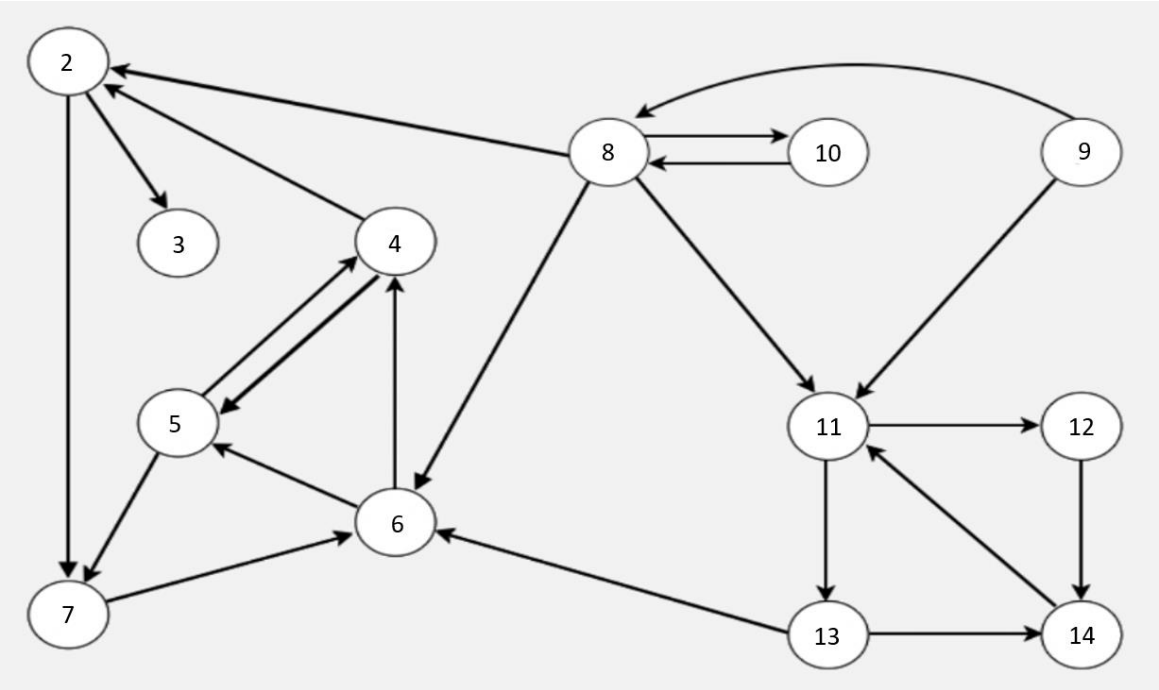
1. Range search is the most important advantage of BST over hashtable. For instance, if one needs to search for elements in certain range one needs to loop over all the elements of hashtable and look up for that value whereas in BST there is systematic way to do this search because of the property of BST that left child of root is always lesser than the root and right child is always greater than the root.

2. In case of large data, it is always better to use BST because as the data grows BST size also increases smoothly unlike hashtable where entire large data is stored at one time unless some algorithms are applied to make the hashtable grow phase-wise.

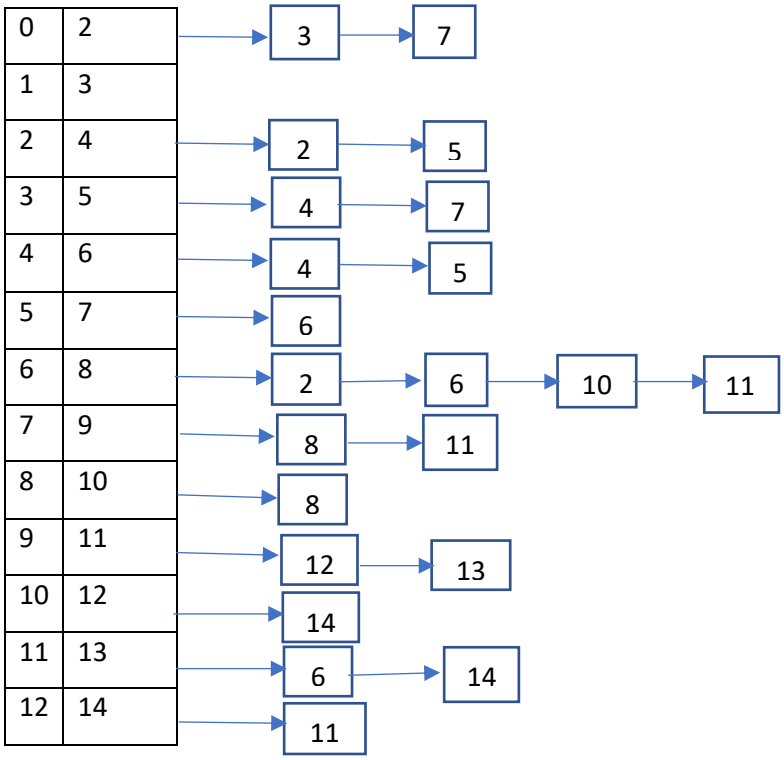
3. We can get all keys in sorted manner by doing inorder traversal of BST. In hashtable, it requires extra processing to sort the elements.

4. Its relatively easy to implement BST than hashtable. This is because hashtable requires to compute hash value for storing and retrieving data. Such calculations may be difficult to write.

5. Hashtable are expected to handle collision whereas in BST no additional processing is required.



1. Write an adjacency list representation of the graph.



2. Perform depth-first-search on the graph and start from node 11. Collect the nodes in pre-order

   Output: 11,12,14,13,6,4,2,3,7,5

   Stack:

~~5~~
~~7~~
~~3~~
~~2~~
~~4~~
~~6~~
~~13~~
~~14~~
~~12~~
~~11~~

3. Repeat 2) but start from node 8

   Output: 8,2,3,7,6,4,5,10,11,12,14,13

   Stack:

~~13~~
~~14~~
~~12~~
~~11~~
~~10~~
~~5~~
~~4~~
~~6~~
~~7~~
~~3~~
~~2~~
~~8~~

4. Perform breadth-first-search on the graph and start from node 11

   Queue :

~~11~~	~~12~~	~~13~~	~~14~~	~~6~~	~~4~~	~~2~~	~~3~~	~~7~~

   Output: 11, 12, 13, 14, 6, 4, 2, 3, 7

Problem 3

- Explain what a collision in hash tables is.

In hashtable data structure, the data elements need to be stored are based on the keys generated using a hash function. The same hash function is also used to search or delete that element.

By applying hash functions on the keys, hashkeys are generated and sometimes these result in calculating the same hashKeys for different elements. In such scenario, two or more elements qualify to be mapped to same location in the hashtable. Such situation is known as **collision**.

- Explain two techniques that can solve the collision issue.

For collision resolution few techniques are available:

a) One such technique is called ***Open Addressing***:

In this approach when collision happens one need to find an open slot where this element can be stored. Simple way would be to start with the location calculated by the hash function and then move sequentially until a vacant slot is found. This is called ***Linear probing***.

For example, if an element say 90 has hashkey of 4, the algorithm will check whether the spot at index=4 is free or not. If its occupied, then as per linear probing approach next index is checked. Index=5 is also occupied so next vacant space is searched.

As per this example index=6 is vacant hence 90 will be inserted at index=6.

Index value:

0	1	2	3	4	5	6	7	8	9
22	33			78	79		34		39

After inserting 90:

22	33			78	79	**90**	34		39

b) Another technique is ***Chaining or Open hashing***

In this approach data element is not stored directly at the hashkey index in the hashtable. Instead the value at the hashkey index actually stores the pointer to the head of the data structure that contains the data. The data structure most commonly used here is linked list.

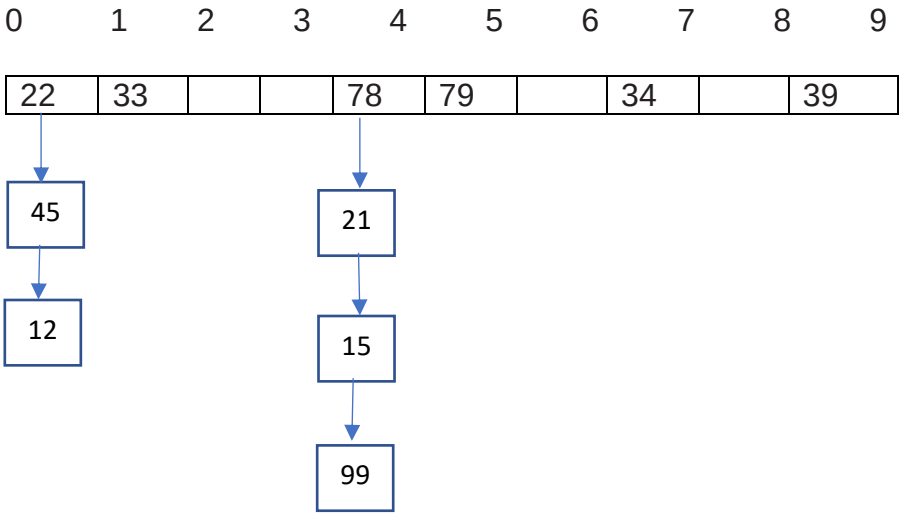
Here all the elements having same hashkey or index are stored in a linked list as depicted below.

In below example: the hashkey/index of 45 and 12 is 0.

Hashkey of 33 is 1.

Hashkey/index of 21,15 and 99 is 4 and so on.

Index value:



References:

- https://brackece.wordpress.com/2012/09/18/hash-table-vs-binary-search-tree/
- https://afteracademy.com/blog/binary-search-tree-vs-hash-table
- https://runestone.academy/runestone/books/published/pythonds/SortSearch/Hashing.html
- https://runestone.academy/runestone/books/published/pythonds/Graphs/BuildingtheWordLadderGraph.html
- https://www.geeksforgeeks.org/implementation-of-hashing-with-chaining-in-python/
- https://www.educative.io/edpresso/what-is-the-word-ladder-problem
- Also referred to many Youtube videos on DFS and BFS