Theory Exam

Name: Jessica Marline Hermawan

NIM: 2440026313

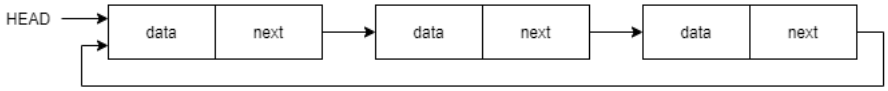
Linked List

1. **Single linked list** is the most common linked list. Each node has data and a pointer to the next node.

In **double linked list** we add a pointer to the previous node in a doubly-linked list. Thus, we can go in either direction: forward or backward.

A **circular linked list** can be either singly linked or doubly linked.

- for singly linked list, next pointer of last item points to the first item
- In the doubly linked list, prev pointer of the first item points to the last item as well.



2. In the array the elements belong to indexes, if you want to get into the fourth element you have to write the variable name with its index or location within the square bracket while in a linked list you have to start from the head and work your way through until you get to the fourth element. Accessing an element in an array is fast, while Linked list takes linear time, so it is quite a bit slower.

3. The **Floyd Warshall algorithm**, itis the algorithm in which there is the use of different characterization of structure for a shortest path that we used in the matrix multiplication which is based on all pair algorithms. In the **Floyd Warshall algorithm**, there are many ways for the constructing the shortest paths. One way is to compute the matrix D of the shortest path weights and then construct the predecessor matrix π from the matrix D. This method can be implemented to run in O (n ^ 3) time.

Pseudocode:

$$Floyd - Warshall(W)$$
$$n = W.rows$$
$$D^{(0)} = W$$
$$for\ k = 1\ to\ n$$
$$let\ D^{(k)} = \left(d_{ij}^{k}\right) be\ a\ new\ n\ \times n\ matrix$$
$$for\ i = 1\ to\ n$$
$$for\ j = 1\ to\ n$$
$$d_{ij}^{(k)} = min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$
$$return\ D^{(n)}$$

Stack and Queue

1. Stack is internally implemented in such a way that the element inserted at the last in stack would be the first element to come out of it. So stack follows LIFO (Last in and First out). On other hand Queue is implemented in such manner that the element inserted at the first in queue would be the first element to come out of it. So queue follows FIFO (First in and First out).

   In case of Stack operations on element take place only from one end of the list called the top. In case of Queue operations on element i.e insertion takes place at the rear of the list and the deletion takes place from the front of the list.

   In Stack operations termed as Push and Pop. While in case of Queue operations termed as Enqueue and dequeue.

2. Infix notation: X + Y

   Operators are written in-between their operands. This is the usual way we write expressions. An expression such as `A * ( B + C ) / D` is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

   Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets `( )` to allow users to override these rules. For example, the usual rules for associativity say that we perform operations from left to right, so the multiplication by A is assumed to come before the division by D. Similarly, the usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction.

Postfix notation (also known as "Reverse Polish notation"): X Y +

Operators are written after their operands. The infix expression given above is equivalent to `A B C + * D /`

The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication.

Operators act on values immediately to the left of them. For example, the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit:

```
( (A (B C +) *) D /)
```

Thus, the "*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

Prefix notation (also known as "Polish notation"): + X Y

Operators are written before their operands. The expressions given above are equivalent to `/ * A + B C D`

As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear:

```
(/ (* A (+ B C) ) D)
```

Although Prefix "operators are evaluated left-to-right", they use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication).

Because Postfix operators use values to their left, any values involving computations will already have been calculated as we go left-to-right, and so the order of evaluation of the operators is not disrupted in the same way as in Prefix expressions.

Infix	Postfix	Prefix	Notes
`A * B + C / D`	`A B * C D / +`	`+ * A B / C D`	multiply A and B, divide C by D, add the results
`A * (B + C) / D`	`A B C + * D /`	`/ * A + B C D`	add B and C, multiply by A, divide by D
`A * (B + C / D)`	`A B C D / + *`	`* A + B / C D`	divide C by D, add B, multiply by A

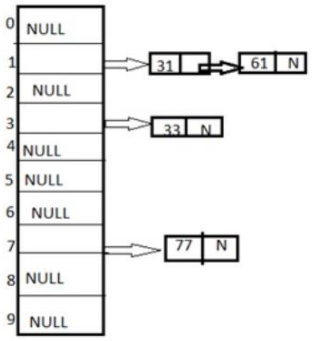
Hashing and Hash Tables

1. **Hash table** is a type of data structure which is used for storing and accessing data very quickly. Insertion of data in a table is based on a key value. Hence every entry in the hash table is defined with some key. By using this key data can be searched in the hash table by few key comparisons and then searching time is dependent upon the size of the hash table.

   Hash table is a type of data structure which is used for storing and accessing data very quickly. Insertion of data in a table is based on a key value. Hence every entry in the hash table is defined with some key. By using this key data can be searched in the hash table by few key comparisons and then searching time is dependent upon the size of the hash table.

   Collision is a situation in which the hash function returns the same hash key for more than one record, it is called as collision. Sometimes when we are going to resolve the collision it may lead to a overflow condition and this overflow and collision condition makes the poor hash function.

2. **Chaining**
   It is a method in which additional field with data i.e. chain is introduced. A chain is maintained at the home bucket. In this when a collision occurs then a linked list is maintained for colliding data.

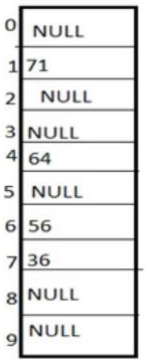


   Example:
   Let us consider a hash table of size 10 and we apply a hash function of H(key)=key % size of table. Let us take the keys to be inserted are 31,33,77,61. In the above diagram we can see at same bucket 1 there are two records which are maintained by linked list or we can say by chaining method.

Linear Probing

It is very easy and simple method to resolve or to handle the collision. In this collision can be solved by placing the second record linearly down, whenever the empty place is found. In this method there is a problem of clustering which means at some place block of a data is formed in a hash table.

Example: Let us consider a hash table of size 10 and hash function is defined as H(key)=key % table size. Consider that following keys are to be inserted that are 56,64,36,71.



In this diagram we can see that 56 and 36 need to be placed at same bucket but by linear probing technique the records linearly placed downward if place is empty i.e. it can be seen 36 is placed at index 7.

Binary Search Tree

1. A **binary tree** is a tree-type non-linear data structure with a maximum of two children for each parent. Every node in a **binary tree** has a left and right reference along with the data element. The node at the top of the hierarchy of a tree is called the root node. The nodes that hold other sub-nodes are the parent nodes.

   A parent node has two child nodes: the left child and right child. Hashing, routing data for network traffic, data compression, preparing binary heaps, and binary search trees are some of the applications that use a binary tree.

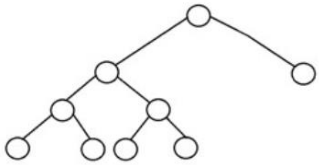
- **Node:** It represents a termination point in a tree.
- **Root:** A tree's topmost node.
- **Parent:** Each node (apart from the root) in a tree that has at least one sub-node of its own is called a parent node.
- **Child:** A node that straightway came from a parent node when moving away from the root is the child node.
- **Leaf Node:** These are external nodes. They are the nodes that have no child.
- **Internal Node:** As the name suggests, these are inner nodes with at least one child.
- **Depth of a Tree:** The number of edges from the tree's node to the root is.
- **Height of a Tree:** It is the number of edges from the node to the deepest leaf. The tree height is also considered the root height.

Binary tree types:

1. Full binary tree

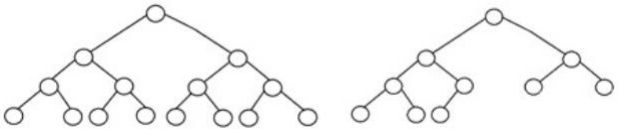
   It is a special kind of a binary tree that has either zero children or two children. It means that all the nodes in that binary tree should either have two child nodes of its parent node or the parent node is itself the leaf node or the external node.

   In other words, a full binary tree is a unique binary tree where every node except the external node has two children. When it holds a single child, such a binary tree will not be a full binary tree. *Here, the quantity of leaf nodes is equal to the number of internal nodes plus one. The equation is like* L=I+1, where L is the number of leaf nodes, and I is the number of internal nodes.



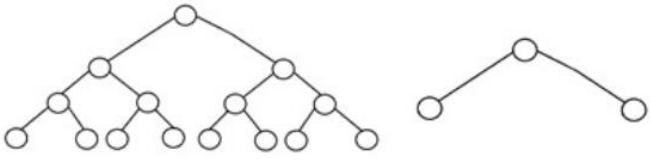
2. Complete binary tree

   A complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also, in the last or the lowest level of this binary tree, every node should possibly reside on the left side.



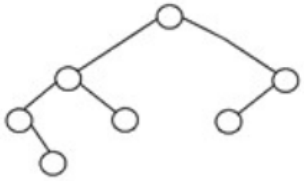
3. Perfect binary tree

   A binary tree is said to be ‘perfect’ if all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree. A perfect binary tree having height ‘h’ has 2h – 1 node. Here is the structure of a perfect



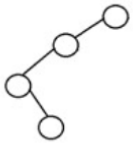
4. Balanced binary tree

   A binary tree is said to be ‘balanced’ if the tree height is O(logN), where ‘N’ is the number of nodes. In a balanced binary tree, the height of the left and the right subtrees of each node should vary by at most one. An AVL Tree and a Red-Black Tree are some common examples of data structure that can generate a balanced binary



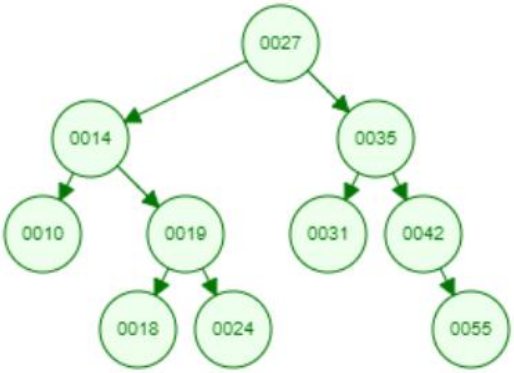
5. Degenerate binary tree

   A binary tree is said to be a degenerate binary tree or pathological binary tree if every internal node has only a single child. Such trees are similar to a linked list performance-wise.



2. Simulate and explain clearly step by step the process of insertion: 24, 18, 55!

- 24 < 35 so it goes to the left branch, and because 31 is bigger than 24 therefore 24 become the child of 19.
- 18 < 19 therefore it become the child of 19
- 55 > 35 so it goes to the right branch, 31 < 42 so 55 become the child of 42 and because 55 is the current largest number in the tree



3. Simulate and explain clearly step by step the process of deletion: 27, 35, 42!

- After 27 the next smaller number is 24, therefore 24 become root node in level 0 in the tree.
- 31 and 42 is the child of 35, therefore it will search for the smaller number which is 31. So when 35 is deleted 31 will replace 35.
- 42 > 24 so it look at right subtree, 42 > 31 so it keep looking at right subtree, when 42 is found, 42 is deleted. Node to delete has no left child so set parent of deleted node to right child of deleted node.