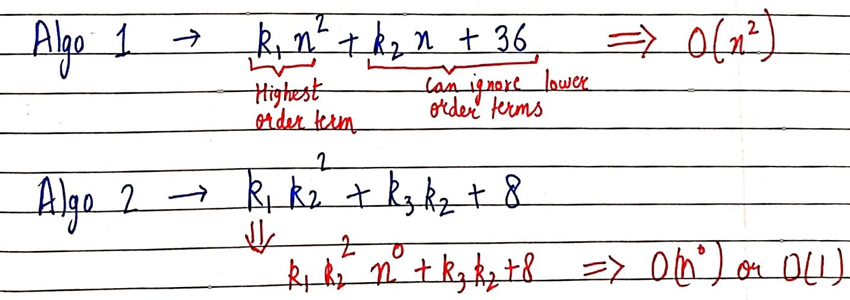
DSA

1. Time Complexity

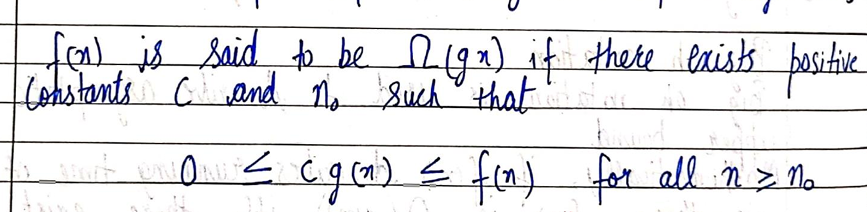


* In order to find big O of an function
* We need to first consider highest order term (in this case (n^2) and can ignore others (in this case (k2 n+36)
* Then removing the constant beside the highest order term and adding BIG O with it.

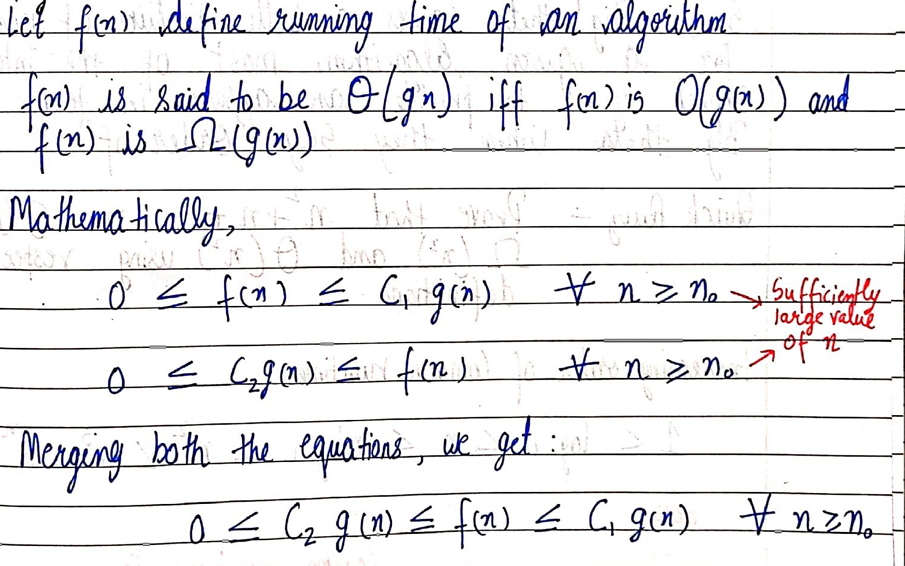
BIG O



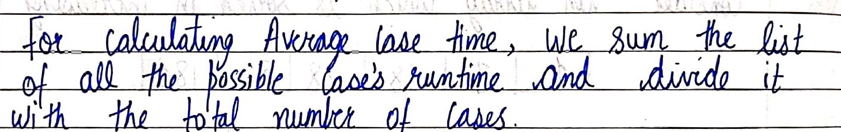
BIG OMEGA

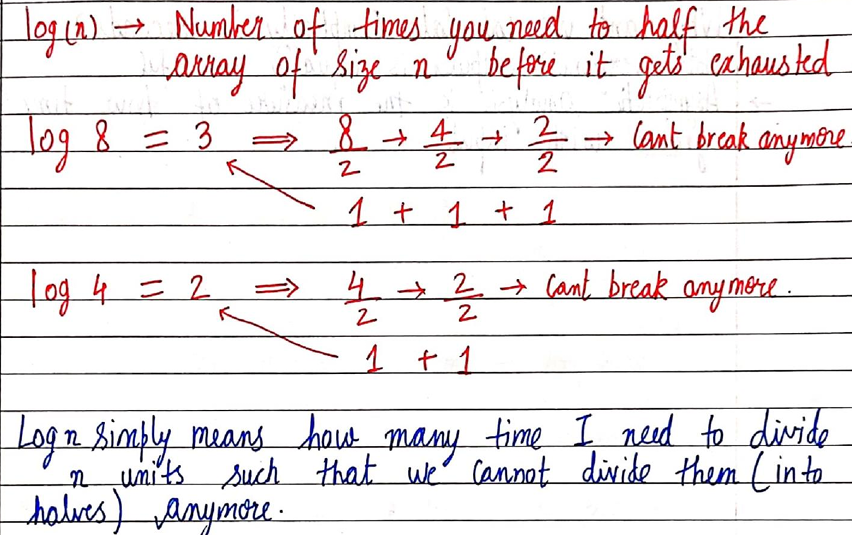


BIG THETA

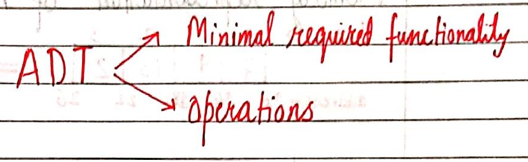


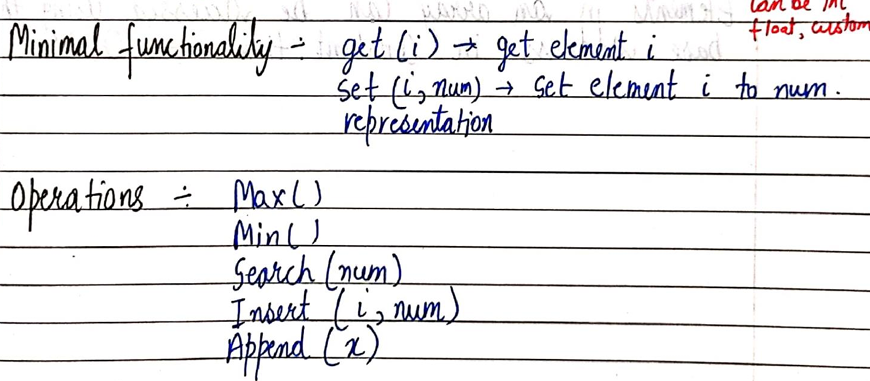
* BEST , WORST and AVERAGE CASE TIME

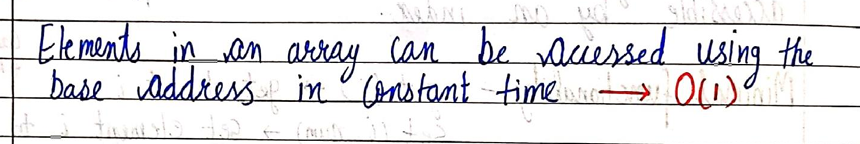




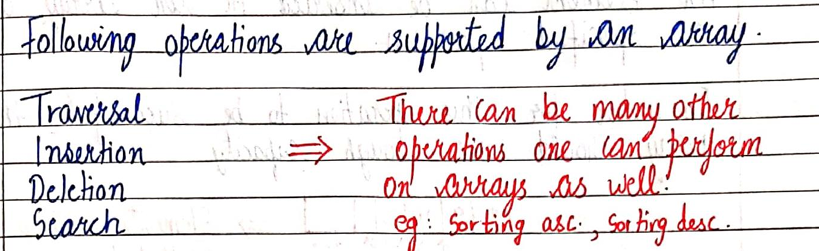
1. ADT

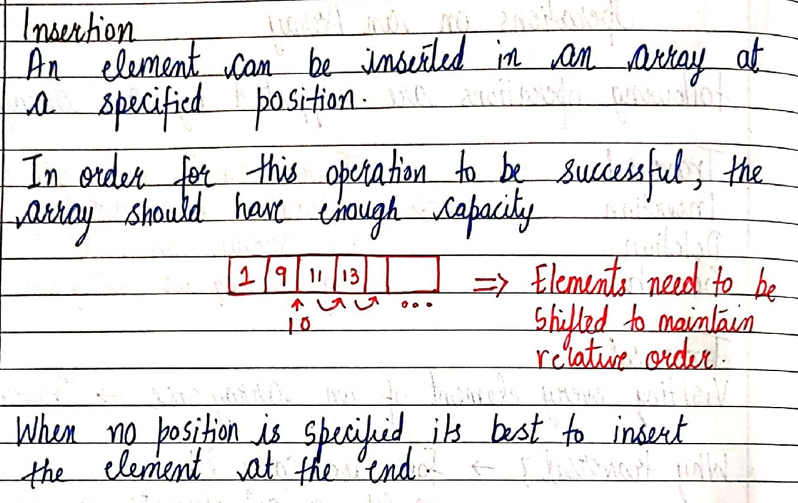




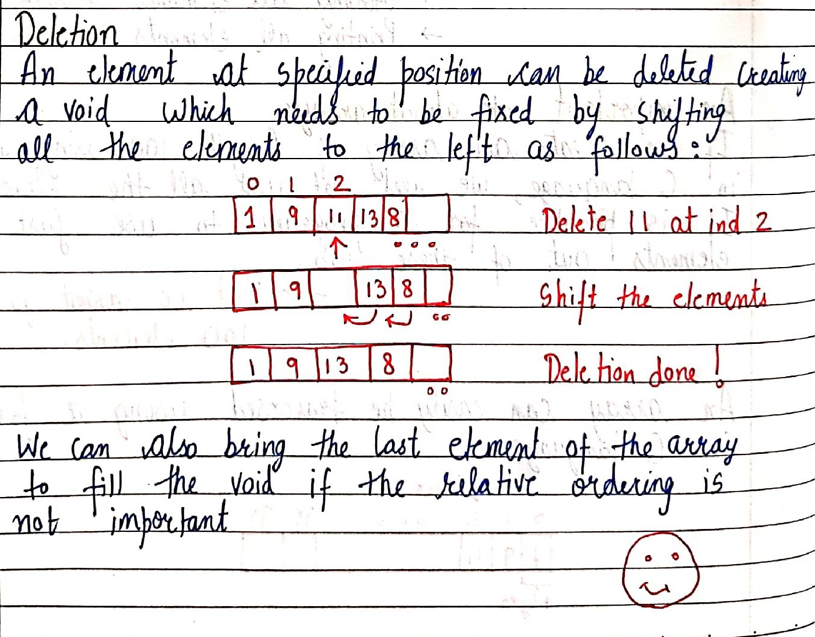


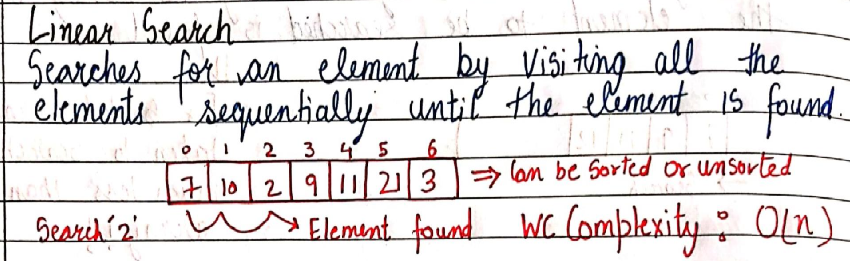
1. Operations in Array

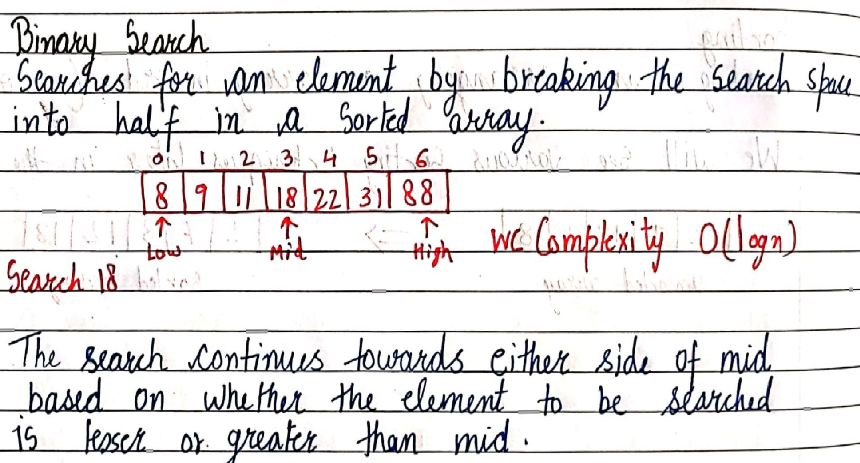


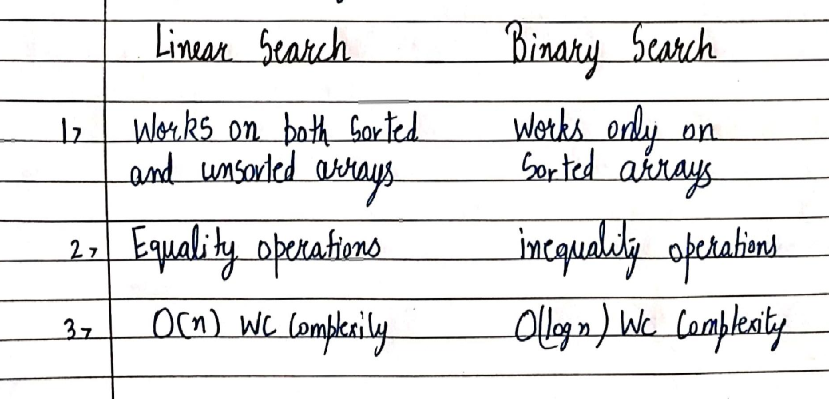


<https://github.com/HK51104/self-study/blob/main/insertion.c>

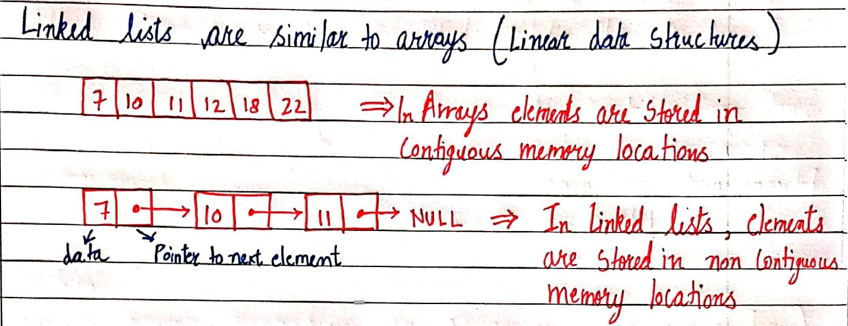


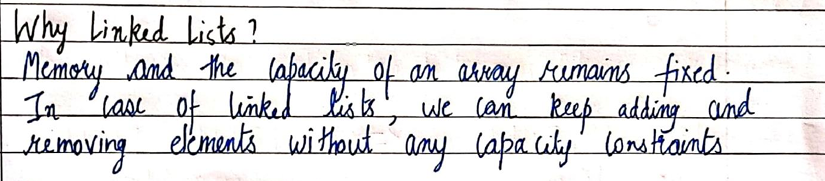


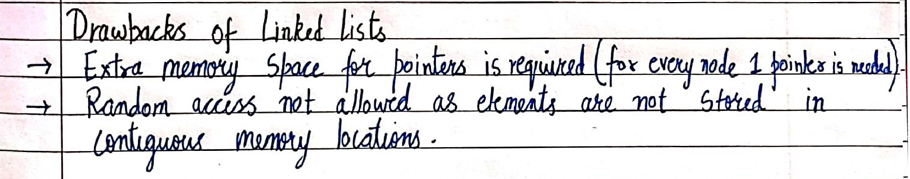




1. LINKED LISTS









This statement means that in order to access the 5 th element of the linked list we need to go through all the 4 elements before it ,unlike array, wherein we can directly access any element as long as we know the index/Position of the element

1. INSERTION AND DELETION IN LINKED LIST

<https://github.com/HK51104/self-study/blob/main/linkedlistinsertion.c>

LINKED LIST

NULL

DATA

POINTER

DATA

DATA

POINTER

POINTER

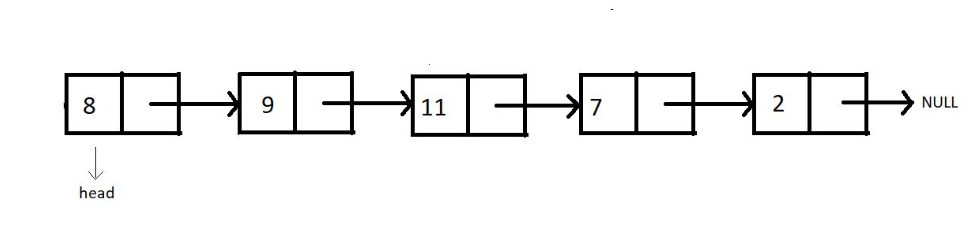
DATA

HEAD

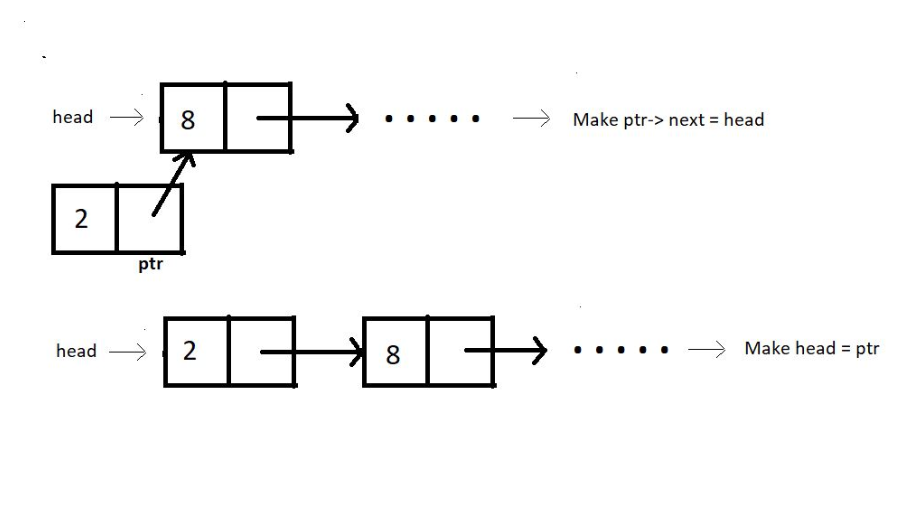
NULL because last member of linked list

points to NULL

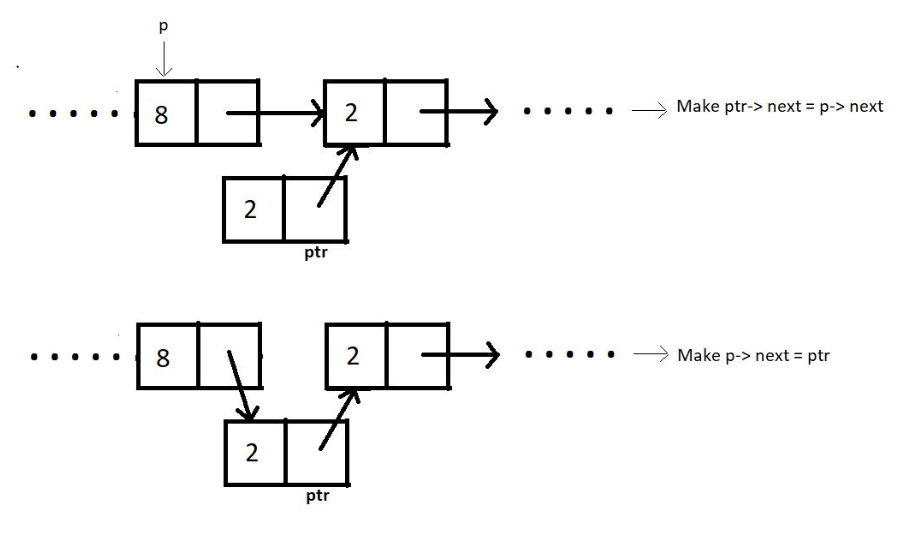
LINKED LIST



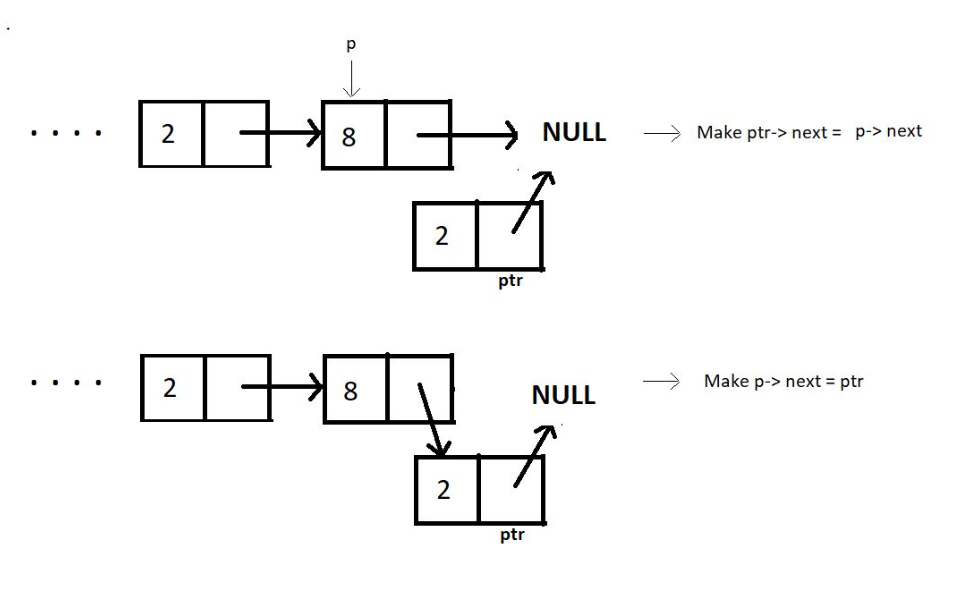
INSERTION AT BEGINNING



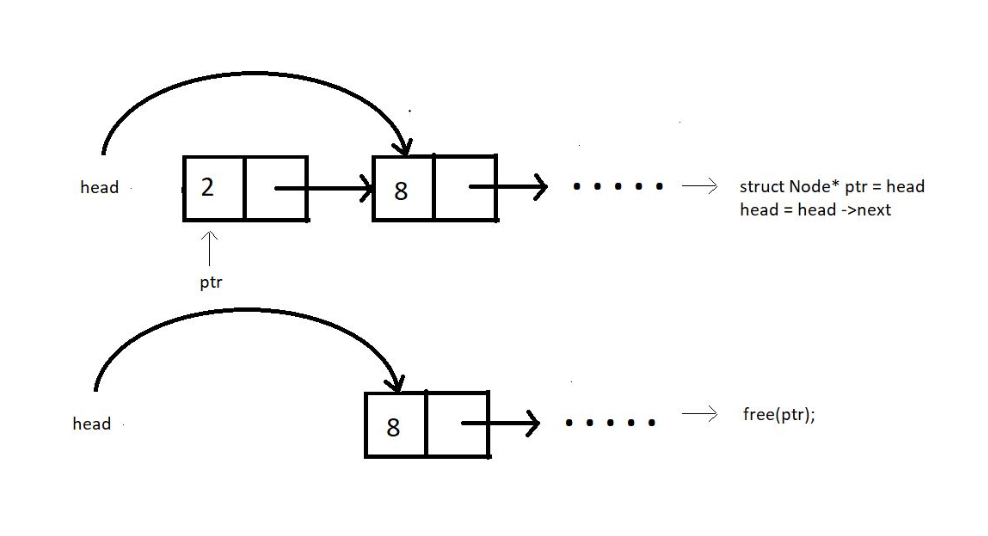
INSERTION IN BETWEEN



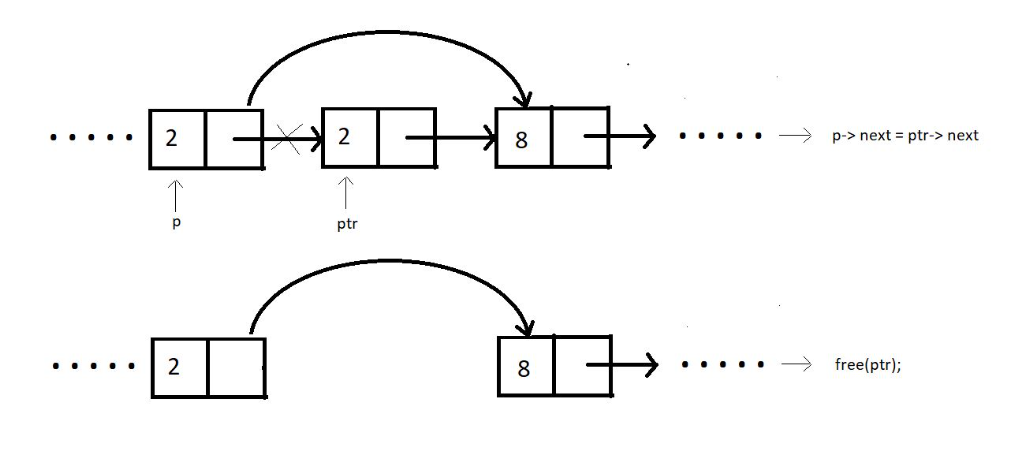
INSERT AT END



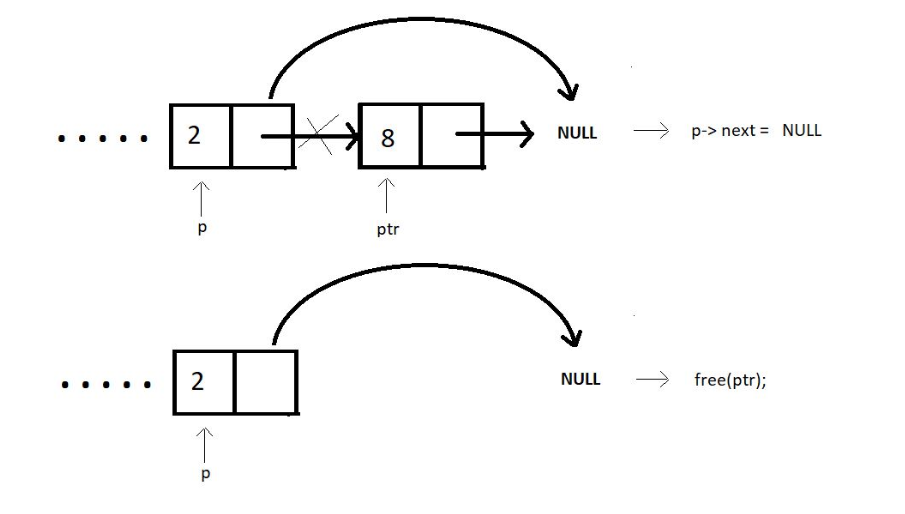
DELETE AT BEGINNING



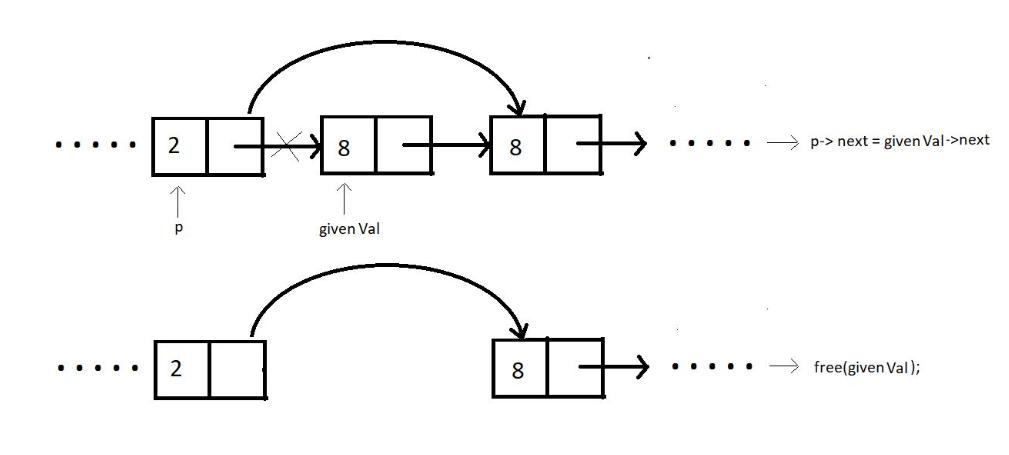
DELETE IN BETWEEN



DELETE AT END

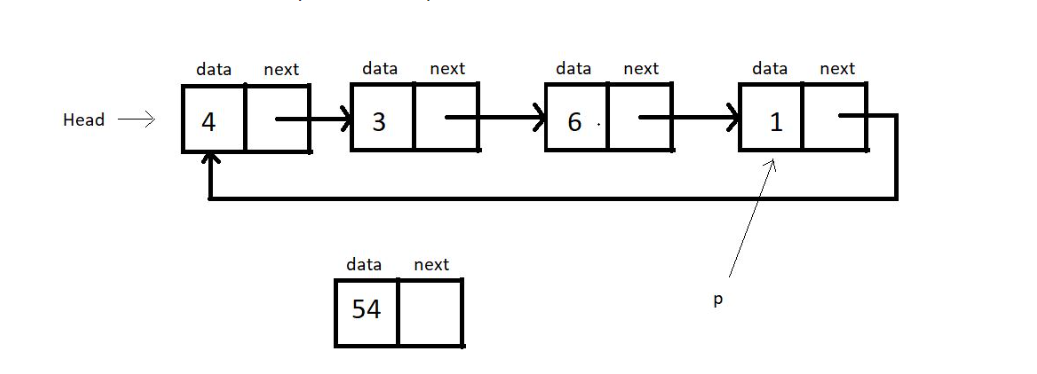


DELETE FOR ANY GIVEN VALUE



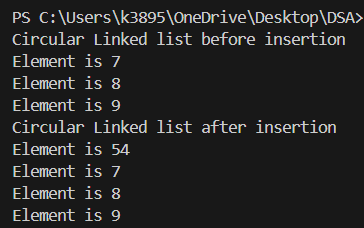
1. CIRCULAR LINKED LIST

* Unlike singly-linked lists, a circular linked list has no node pointing to NULL. Hence it has no end. The last element points at the head node.



<https://github.com/HK51104/self-study/blob/main/circularlinkedlist.c>

OUTPUT:



1. DOUBLY LINKED LIST

Each node contains a data part and two pointers in a doubly-linked list, one for the previous node and the other for the next node.

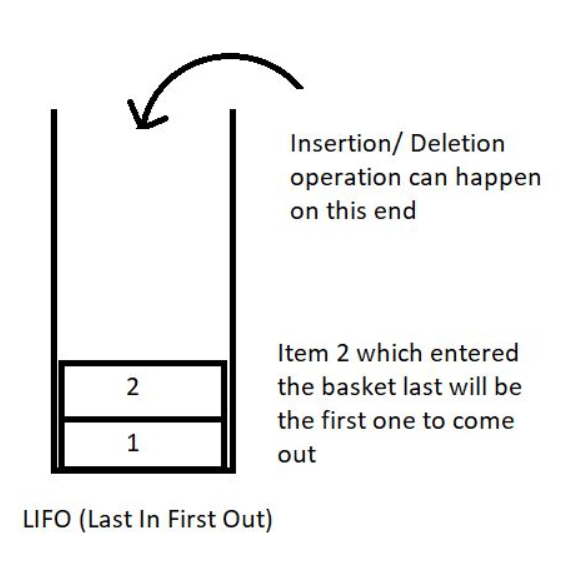


 Both the end pointers point to the NULL.

* A doubly linked list allows traversal in both directions. We have the addresses of both the next node and the previous node. So, at any node, we’ll have the freedom to choose between going right or left.

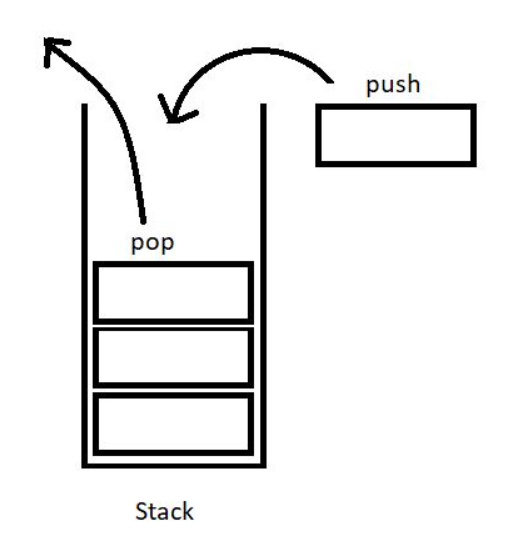
1. STACK

Any operation on the stack is performed in LIFO (Last In First Out) order. This means the element to enter the container last would be the first one to leave the container.



Here are some of the basic operations we would want to perform on stacks:

1. push(): to push an element into the stack
2. pop(): to remove the topmost element from the stack
3. peek(index): to return the value at a given index
4. isempty() / isfull() : to determine whether the stack is empty or full to carry efficient push and pull operations.



<https://github.com/HK51104/self-study/blob/main/stack.c>

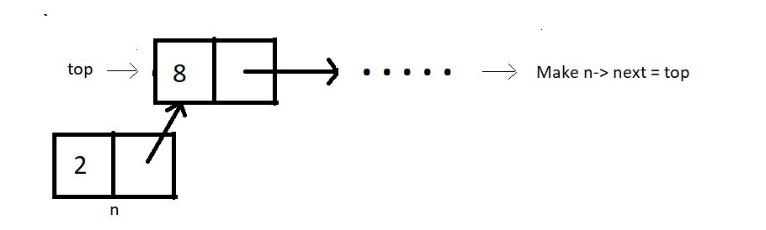
/////////

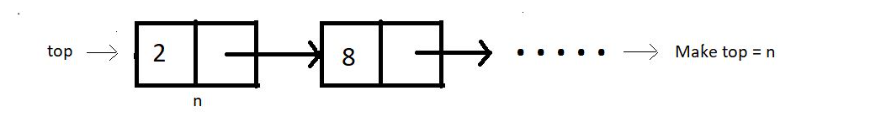
<https://github.com/HK51104/self-study/blob/main/stackimplementation.c>

1. STACK IMPLEMENTATION USING LINKED LIST

<https://github.com/HK51104/self-study/blob/main/stackusinglinkedlist.c>

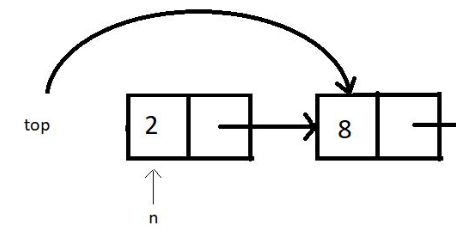




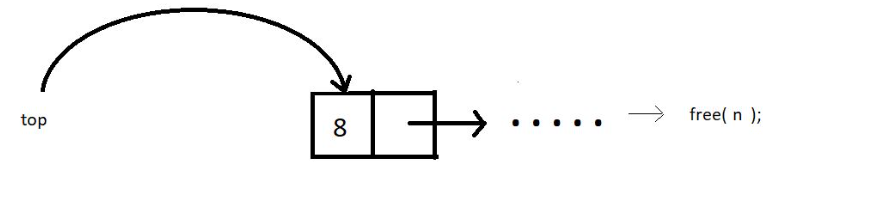




//POP









1. PARENTHESIS MATCHING

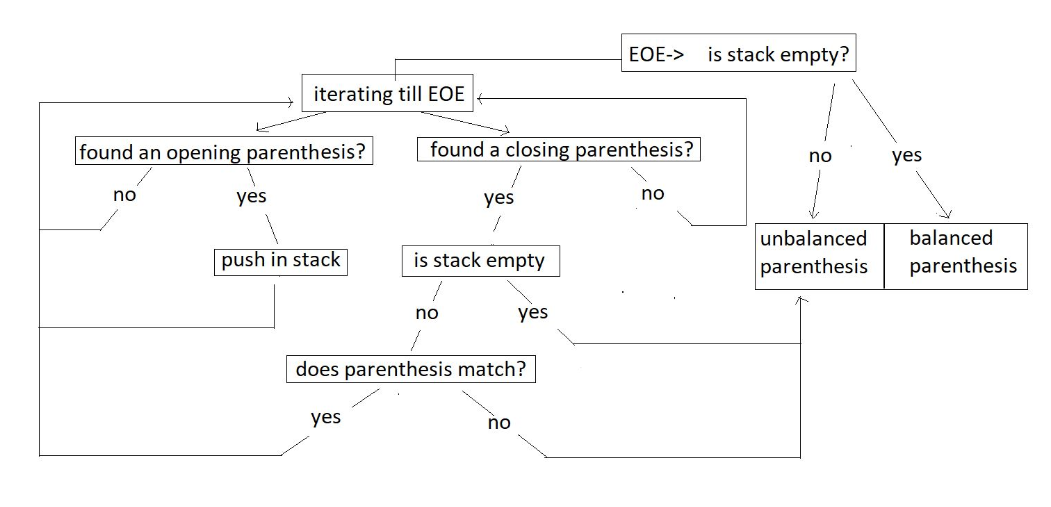
we tried making parentheses matching intuitive and more understandable using stacks. We followed one simple algorithm to accomplish that.

The algorithm states:

* Everytime you come across an opening parenthesis, push it in the stack.
* Everytime you come across a closing parenthesis, pop one opening parenthesis out from the stack.
* We call this match of parentheses unbalanced when we encounter either of the two of these troubles:

1. There is no more opening bracket inside the stack to pop, and you come across a closing bracket.
2. The stack size is not zero, or there are still more than zero opening brackets present in the stack after you come across EOE(end-of-expression).

<https://github.com/HK51104/self-study/blob/main/parenthesismatching.c>



<https://github.com/HK51104/self-study/blob/main/multipleparenthesismatching.c>

1. INFIX,PREFIX,POSTFIX

Infix : A\*(B+C)\*D

Postfix: ABC+\*D

[first the computer try ro reach an “operand”

When it reaches and “operand”

In this case “+” then

It will do the “operation” as mentioned by the

“OPERAND” to the previous two variables

In this case B and C(B+C)

Then it will go to the next “operand”

And do the “operation” as mentioned by the

“operand” to previous two “variables”

In this case A\*(B+C) and so on…

**Infix:**

This is the method we have all been studying and applying for all our academic life. Here the operator comes in between two operands. And we say, two is added to three. For eg: 2 + 3, a \* b, 6 / 3 etc.

< operand 1 >< **operator** >< operand2 >

**Prefix:**

This method might seem new to you, but we have vocally used them a lot as well. Here the operator comes before the two operands. And we say, Add two and three. For e.g.:  + 6 8, \* x y, -  3 2 etc.

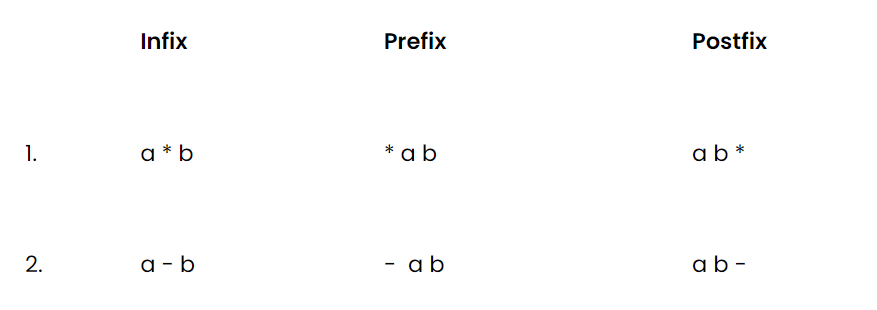
<**operator**>< operand 1 >< operand2 >

**Postfix:**

This is the method that might as well seem new to you, but we have used even this in our communication. Here the operator comes after the two operands. And we say, Two and three are added. For e.g.:  5 7 +, a  b \*,  12 6 / etc.

< operand 1 >< operand2 >< **operator** >

NOTE: Machine finds it easy to evaluate Postfix



**Converting infix to prefix:**

Consider the expression, **x - y \* z**.

1. Parentheses the expression. The infix expression must be parenthesized by following the operator precedence and associativity before converting it into a prefix expression. Our expression now becomes **( x - ( y \* z ) )**.

2. Reach out to the innermost parentheses. And convert them into prefix first, i.e.  **( x - ( y \* z ) )**changes to **( x - [ \* y z ] )**.

3. Similarly, keep converting one by one, from the innermost to the outer parentheses.  **( x - [ \* y z ] )  → [ - x \* y z ].**

4. And we are done.

**Converting infix to postfix:**

Consider the same expression, **x - y \* z**.

5. Parentheses the expression as we did previously. Our expression now becomes **( x - ( y \* z ) )**.

6. Reach out to the innermost parentheses. And convert them into postfix first, i.e.  **( x - ( y \* z ) )**changes to **( x - [ y z \* ] )**.

7. Similarly, keep converting one by one, from the innermost to the outer parentheses.  **( x - [ y z \* ] )  → [ x y z \* - ].**

8. And we are done.

Similarly the expression p - q -  r / a, follows the following conversions to become a prefix expression:

* **p - q -  r / a**  →  ( ( p - q ) -  ( r / a ) ) →  ( [ - p q ] - [ / r a ]  )  →**- - p q / r a**

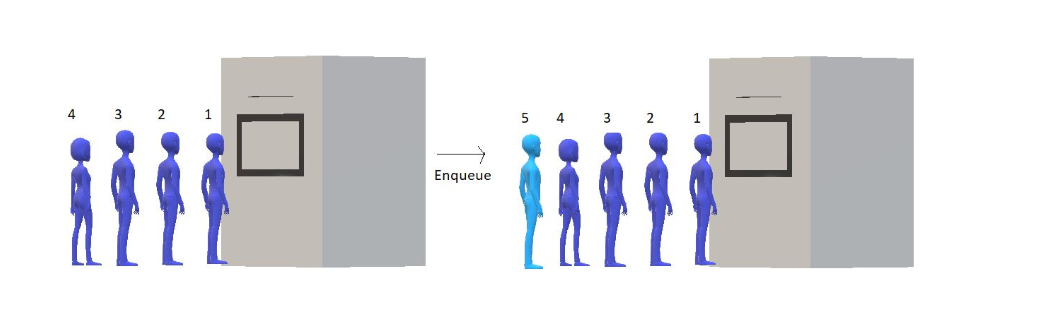
<https://github.com/HK51104/self-study/blob/main/infixtopostfix.c>

1. Queue

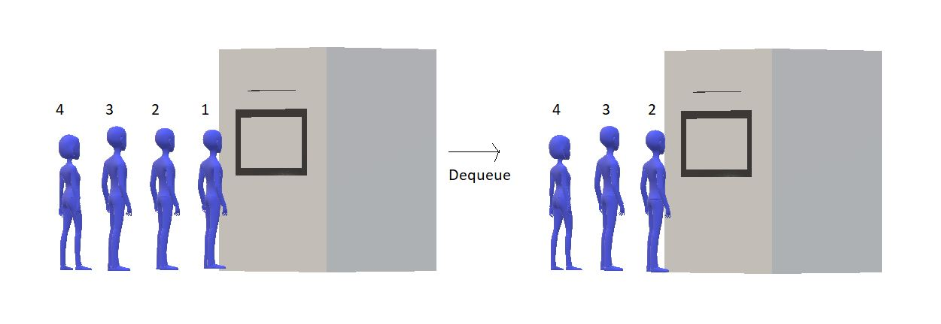
Unlike stacks, where we followed LIFO( Last In First Out ) discipline, here in the queue, we have FIFO( First In First Out).

In stacks, we had to maintain just one end, *head,* where both insertion and deletion used to take place, and the other end was closed. But here, in queues, we have to maintain both the ends because we have insertion at one end and deletion from the other end.

* 1. enqueue() : to insert an element in a queue.



* 1. dequeue(): to remove an element from the queue



<https://github.com/HK51104/self-study/blob/main/queue.c>

NOTE: Whenever a “structure” is made

It is meant that a “block of space” is being allocated to that “structure”

The “size” of “block of space” depends on the “data types” present inside the “structure”

struct queue

{

    int size;

    int f;

    int r;

    int \*arr;

};

    struct queue q;

This basically means that

“queue” name ka “structure” banaya gaya hai

And “q” is one “variable” of “data type” “struct queue”

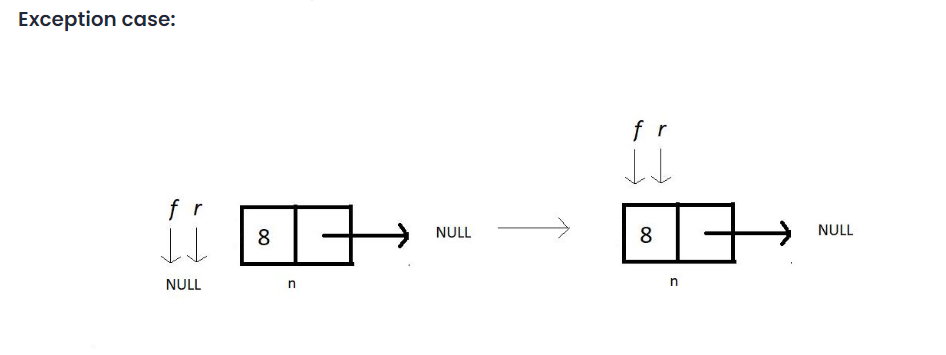
Which further has “many data types”

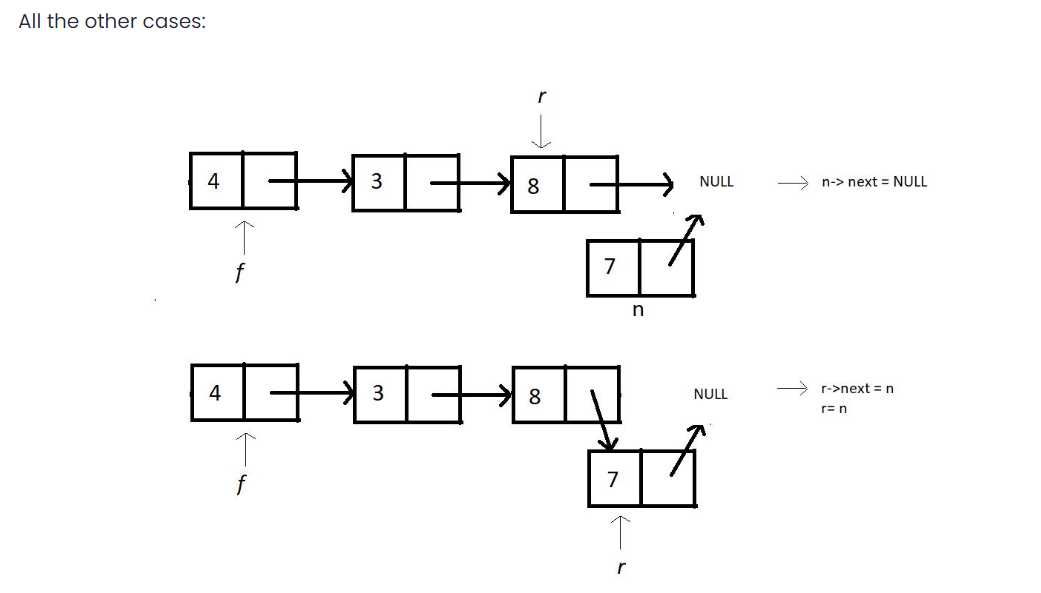
1. CIRCULAR QUEUE

<https://github.com/HK51104/self-study/blob/main/circularqueue.c>

1. QUEUE USING LINKED LIST

<https://github.com/HK51104/self-study/blob/main/queueusinglinkedlist.c>





1. DE-QUEUE

//code to be written

1. SORTING

criteria for analyzing different sorting algorithms and why one differs from the other

1. Time Complexity

We observe the time complexity of an algorithm to see which algorithm works efficiently for larger data sets and which algorithm works faster with smaller data sets. What if one sorting algorithm sorts only 4 elements efficiently and fails to sort 1000 elements. What if it takes too much time to sort a large data set? These are the cases where we say the time complexity of an algorithm is very poor.

In general, O(N log N) is considered a better algorithm time complexity than O(N2), and most of our algorithms’ time complexity revolves around these two.

1. Space Complexity

The space complexity criterion helps us compare the space the algorithm uses to sort any data set. If an algorithm consumes a lot of space for larger inputs, it is considered a poor algorithm for sorting large data sets. In some cases, we might prefer a higher space complexity algorithm if it proposes exceptionally low time complexity, but not in general.

And when we talk about space complexity, the term **in-place sorting algorithm**arises. The algorithm which results in constant space complexity is called an in-place sorting algorithm. Inplace sorting algorithms mostly use swapping and rearranging techniques to sort a data set. One example is Bubble Sort

1. Stability

The stability of an algorithm is judged by the fact whether the order of the elements having equal status when sorted on some basis is preserved or not. It probably sounded technical, but let me explain.

Suppose you have a set of numbers, 6, 1, 2, 7, 6, and we want to sort them in increasing order by using an algorithm. Then the result would be 1, 2, 6, 6, 7. But the key thing to look at is whether the 6s follow the same order as that given in the input or they have changed. That is, whether the first 6 still comes before the second 6 or not. If they do, then the algorithm we followed is called stable, otherwise unstable.

1. Internal & External Sorting Algorithms

When the algorithm loads the data set into the memory (RAM), we say the algorithm follows internal sorting methods. In contrast, we say it follows the external sorting methods when the data doesn’t get loaded into the memory.

1. Adaptivity

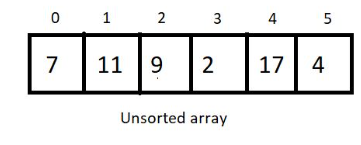
Algorithms that adapt to the fact that if the data are already sorted and it must take less time are called **adaptive algorithms**.  And algorithms which do not adapt to this situation are not adaptive.

1. Recursiveness

If the algorithm uses recursion to sort a data set, then it is called a recursive algorithm. Otherwise, non-recursive.

1. BUBBLE SORT ALGORITHM

With bubble sort, we intend to ensure that the largest element of the segment reaches the last position at each iteration.





Segment

Bubble sort intends to sort an array using (n-1) passes where n is the array's length.



In every iteration, segment will be sorted in such a way that the larger element will take place larger index

And when, the largest element of the current unsorted part reaches its final position

It completes  one pass

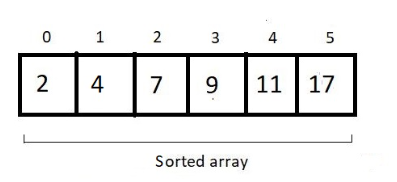
After every pass,

 unsorted part of the array reduces by 1, and the sorted part increases by 1.

At each pass, we will iterate through the unsorted part of the array and compare every adjacent pair. We move ahead if the adjacent pair is sorted; otherwise, we make it sorted by swapping their positions. And doing this at every pass ensures that the largest element of the unsorted part of the array reaches its final position at the end.

Since our array is of length 6, we will make 5 passes.

After 5th pass,



For an array of length n, we would have (n-1) + (n-2) + (n-3) + (n-4) + . . . . . + 1 comparison and possible swaps.

the sum from 1 to n-1, which is n(n-1)/2, and hence our complexity of runtime becomes **O(n^2).**

we never made a swap when two elements of a pair become equal. Hence the algorithm is a **stable algorithm**

This algorithm has no adaptive aspect since every pair will be compared, even if the array given has already been sorted. So, no adaptiveness.

<https://github.com/HK51104/self-study/blob/main/bubble.c>

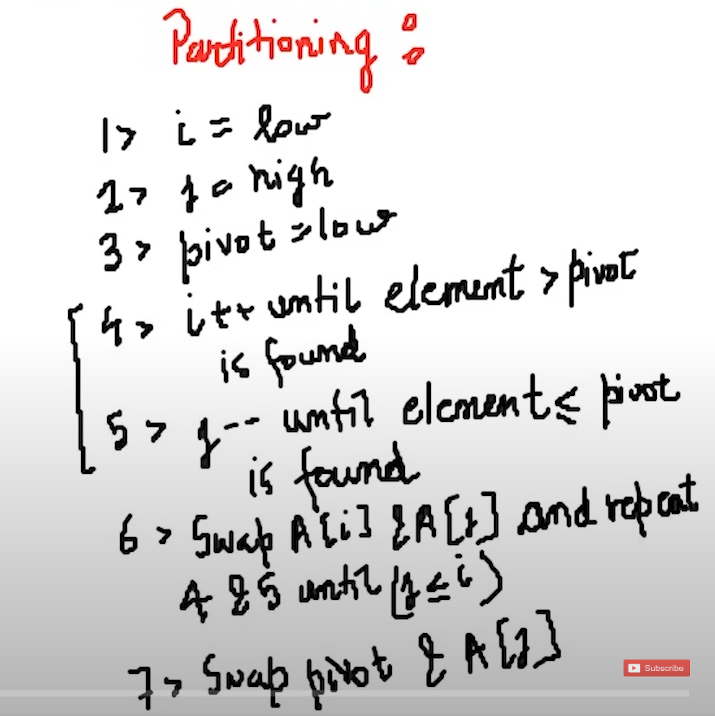
1. Insertion sort

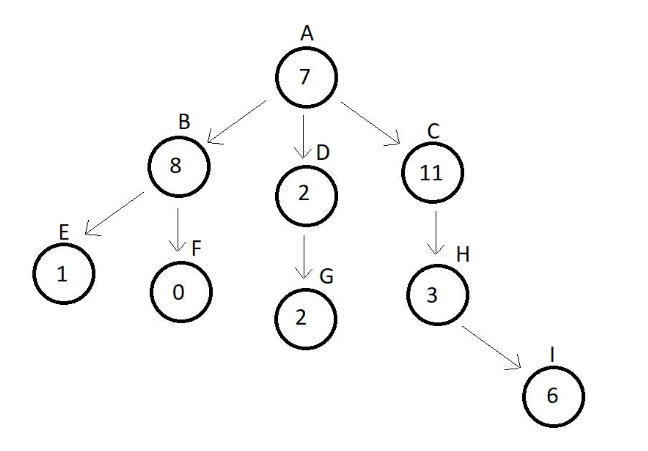
<https://github.com/HK51104/self-study/blob/main/insertionsort.c>

1. Selection sort

<https://github.com/HK51104/self-study/blob/main/selectionsort.c>

1. Quick Sort



1. TREES

**Root:**The topmost node of a tree is called the root. There is no edge pointing to it, but one or more than one edge originating from it.

**Parent:**Any node which connects to the child. Node which has an edge pointing to some other node. Here, C is the parent of H.

**Child:**Any node which is connected to a parent node. Node which has an edge pointing to it from some other node. Here, H is the child of C.

**Siblings:**Nodes belonging to the same parent are called siblings of each other. Nodes B, C and D are siblings of each other, since they have the same parent node A.

**Ancestors:**Nodes accessible by following up the edges from a child node upwards are called the ancestors of that node. Ancestors are also the parents of the parents of …… that node. Here, nodes A, C and H are the ancestors of node I.

**Descendants:**Nodes accessible by following up the edges from a parent node downwards are called the descendants of that node. Descendants are also the child of the child of …… that node. Here, nodes H and I are the descendants of node C.

**Leaf/ External Node:**Nodes which have no edge originating from it, and have no child attached to it. These nodes cannot be a parent. Here, nodes E, F, G and I are leaf nodes.

**Internal node:**Nodes with at least one child. Here, nodes B, D and C are internal nodes.

**Depth:**Depth of a node is the number of edges from root to that node. Here, the depth of nodes A, C, H and I are 0, 1, 2 and 3 respectively.

**Height:**Height of a node is the number of edges from that node to the deepest leaf. Here, the height of node A is 3, since the deepest leaf from this node is node I. And similarly, height of node C is 2.

A tree with**n** nodes has **n-1** edges

**degree of a node** in a tree is the number of children of a node. **degree of a node** in a tree is the number of children of a node.

**degree of a tree** is the highest degree of a node among all the nodes present in the tree.

1. BINARY TREE

A binary tree is a special type of tree where each node has a degree equal to or less than two which means each node should have at most two children.

* **Full or Strict Binary trees:**

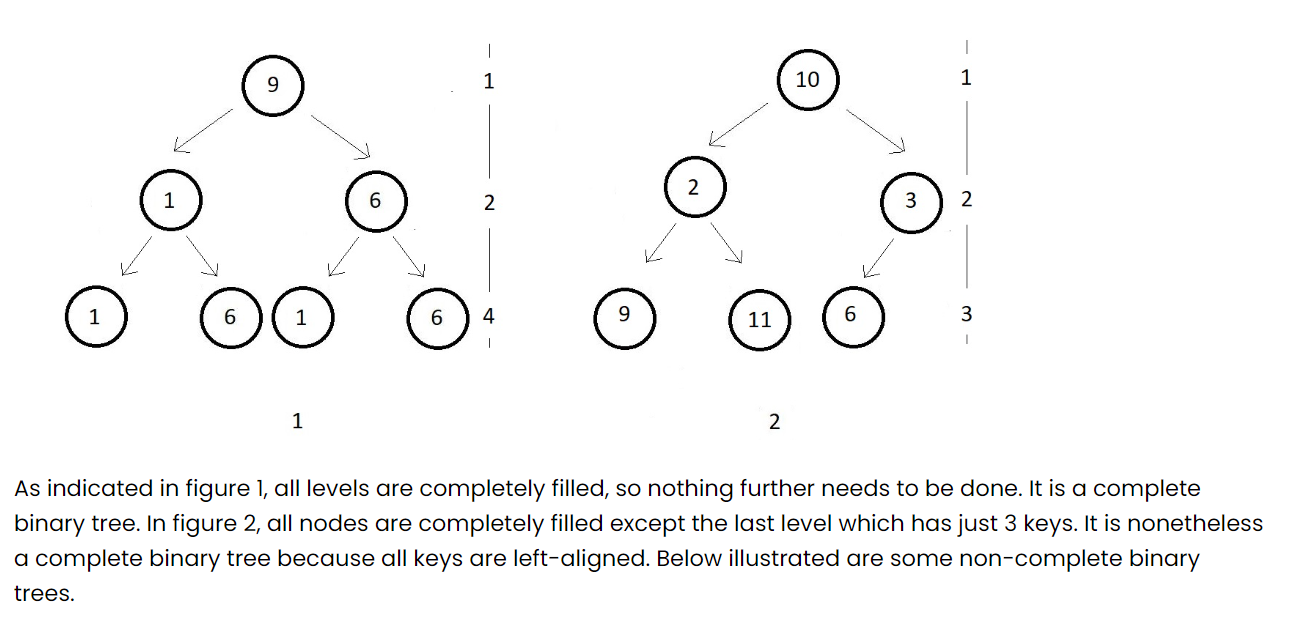
Binary trees as we said earlier have a degree of 2 or less than 2. But a strict binary tree is a binary tree having all of its nodes with a degree of 2 or 0.

* **Perfect Binary Tree:**

A perfect binary tree has all its internal nodes with degree strictly 2 and has all its leaf nodes on the same level.

* **Complete Binary Tree:**

A complete binary tree has all its levels completely filled except possibly the last level. And if the last level is not completely filled then the last level’s keys must be all left-aligned.



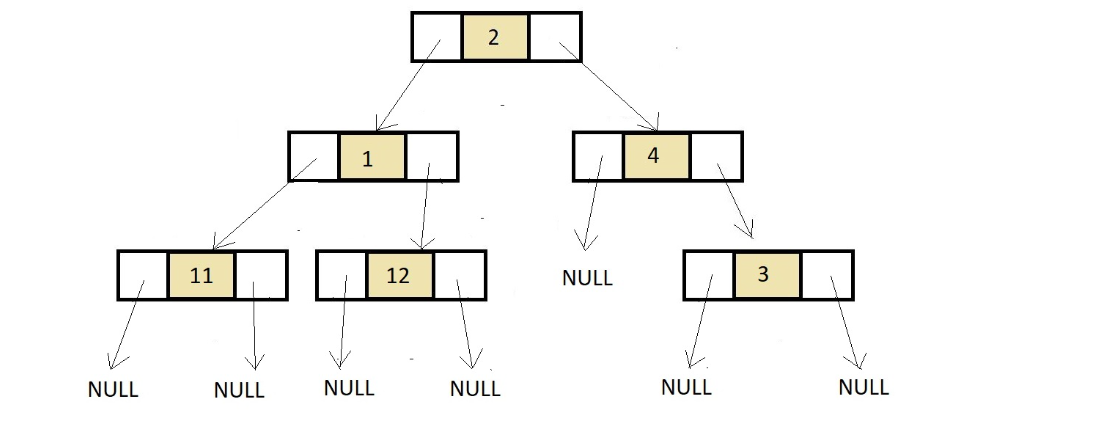
* **Degenerate tree:**

The easiest of all, degenerate trees are binary trees where every parent node has just one child and that can be either to its left or right

* **Skewed trees:**

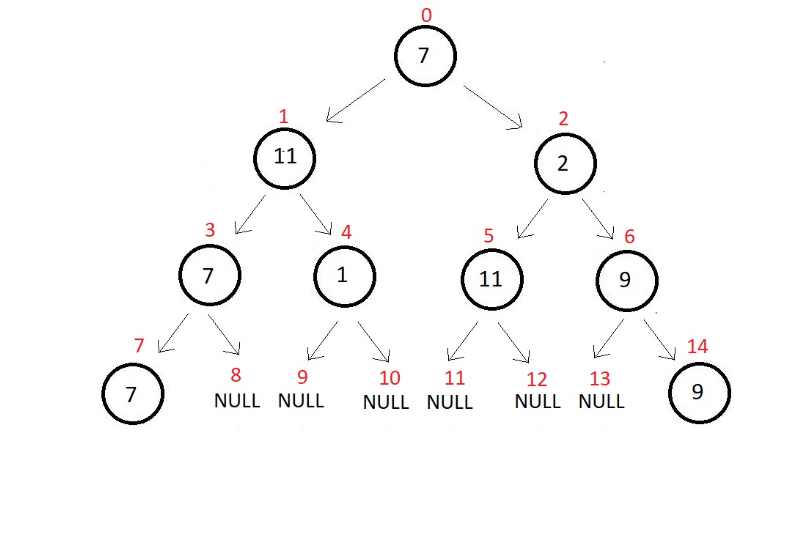
Skewed trees are binary trees where every parent node has just a single child and that child should be strict to the left or to the right for all the parents.

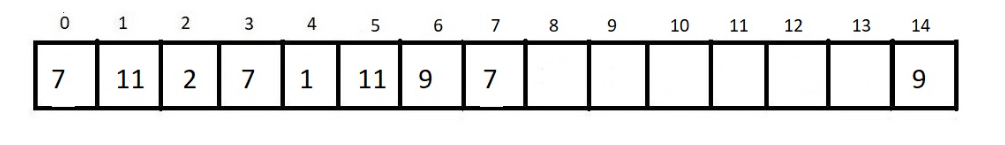
Representation of binary tree is best done sing doubky linked list as



The left pointer stores the data for left side child and the right pointer stores the data for right side child

Whereas in the case of array





1. BINARY TREE REPRESENTATION USING LINKED LIST

Makes it very costly in terms of space which isn’t efficient