Data Structure & Algorithms

Sunbeam Infotech

Module Pre-requisites

- Java Programming Language
  - Language Fundamentals
  - Methods
  - Class & Object
  - static members
  - Arrays
  - Collections
  - Inner classes

1. thinking
2. algorithms
3. implementation
4. interviews

Module plan

1. Time & Space complexity
2. Linear & Binary search
3. Recursion
4. Basic sorting
5. Linked list
6. Queues
7. Stacks
8. Trees
9. Heap
10. Advanced sorting
11. Hashing
12. Graphs

Problem Solving Techniques

1. Divide & Conquer
2. Greedy approach
3. Dynamic Programming

Evaluations

1. Theory - 40 marks - CCEE
2. Lab - 40 marks
3. Interval - 20 marks

Data Structure

- Data Structure
  - Organizing data in memory
  - Processing the data

  } efficiently
- Basic data structures
  - Array
  - Linked List
  - Stack
  - Queue
- Advanced data structures
  - Tree
  - Heap
  - Graph

- Data structures are ADT. ← Object Oriented

Abstract Data Types

Array ADT : random access
1. get ele at index
2. set ele at index

Stack ADT : LIFO
1. push
2. pop
3. peek
4. isEmpty

Queue ADT : FIFO
1. push
2. pop
3. peek
4. isEmpty

Data Structure

- Data Structure
  - Organizing data in memory
  - Processing the data
- Common data structures
  - Array
  - Linked List
  - Stack
  - Queue
- Advanced data structures
  - Tree
  - Heap
  - Graph

efficient

exact analysis
* bytes
* time

asymptotic analysis
* unit space
* iterations

- Asymptotic analysis
  - It is not exact analysis

Big O → Order of
1 → const k
n → S ∝ n, O(n)
n^2 → S ∝ n^2, $O(n^2)$

- Space complexity
  - Unit space to store the data (Input space) and additional space to process the data (Auxiliary space).
  - O(1), O(n), $O(n^2)$

- Time complexity
  - Unit time required to complete any algorithm.
  - Approximate measure of time required to complete any algorithm. → based on some factor.

Time complexity

- Time complexity of a algorithm depends on number of iterations in the algorithm.

- Common time complexities
  - O(1) → T=k
  - O(n) → T∝n → single loop
  - O(n^2) → T∝n^2 → nested loops - (2)
  - O(n^3) → T∝n^3 → nested loops - (3)
  - O(log n) → T∝ log n → partitioning
  - O(n log n) → T∝ n log n → partitioning repeated

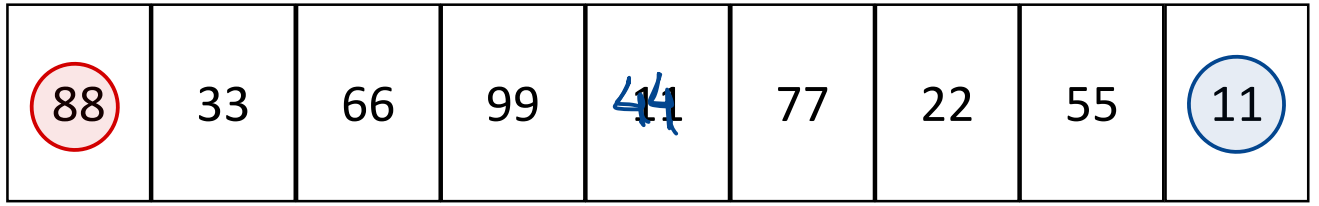
(1) check if num is prime.
→ O(n) :
```
for(i=2; i<n; i++){
    if(n%i ==0)
        pf("not prime");
}
if(i==n)
    pf("prime");
```

(2) print table of "n".
→ O(1) :
```
for(i=1; i<=10; i++)
    pf(num*i);
```

- Asymptotic notations
  - Big O – O(…) – Upper bound → max time - worst case
  - Omega – Ω(…) – Lower bound → min time - best case
  - Theta Θ – Θ(…) – Upper & Lower bound → avg time - avg case

Linear Search

- Find a number in a list of given numbers (random order).




```
for(i=0; i<n; i++) {
    if(a[i]==key)
        return i;   → return index if found
}
return -1;          → if not found.
```

Space Complexity

Input space: $T \propto n$ → $O(n)$

Aux Space: $S = k$ → $O(1)$

- Time complexity
  - Worst case → max iterations (n itr) → $T \propto n$ → $O(n)$
  - Best case → min iterations (only 1 itr.) → $T = k$ → $\Omega(1)$ / $O(1)$
  - Average case → avg iterations → $T \propto \frac{n}{2}$ → $\Theta(n)$ / $O(n)$
    ↳ $T \propto n$

Binary Search

* for sorted array only.

```
while(l <= r)
{
    m = (l+r)/2
    if( key == a[m])
        return m;
    if( key < a[m])
        r = m-1;        <- left part
    else
        l = m+1;        <- right part
}
return -1;
```

0	1	2	3	4	5	6	7	8
11	22	33	44	55	66	77	88	99

R L M

$2^i \simeq n \rightarrow 2^4 = 16$

$i \log 2 = \log n$

$i = \frac{\log n}{\log 2}$

$T \propto i$

$T \propto \frac{\log n}{\log 2}$

$T \propto \log n$

$\rightarrow O(\log n)$

16 → 8 → 4 → 2 → 1

Recursion – Divide & Conquer

- Function calling itself is called as recursive function.
- To write recursive function consider
  - Explain process/formula in terms of itself
  - Decide the end/terminating condition
- Examples:
  - n! = n * (n-1)!
  - $x^y = X * x^{y-1}$
  - $T_n = T_{n-1} + T_{n-2}$
  - factors(n) = prime factor of n * factors(n / prime factor)
- Function call in recursion
- Pros & Cons of recursion

base

0! = 1

$x^0 = 1$

$T_1 = T_2 = 1$

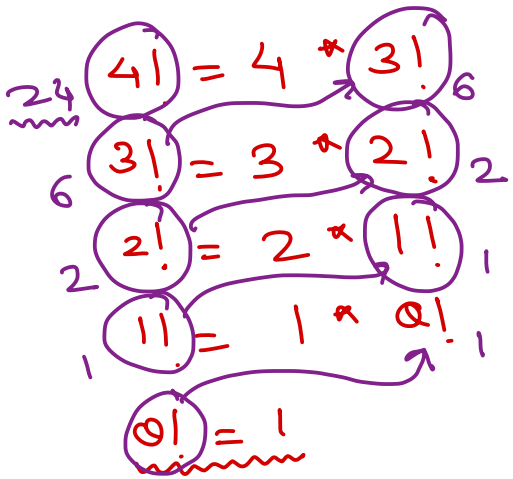
factors(1) = 1

- On each function call, function activation record or stack frame will be created on stack.

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}

res=fact(4);
```

x → return addr.

back-substitution



$2^3 = 2 * 2^2$

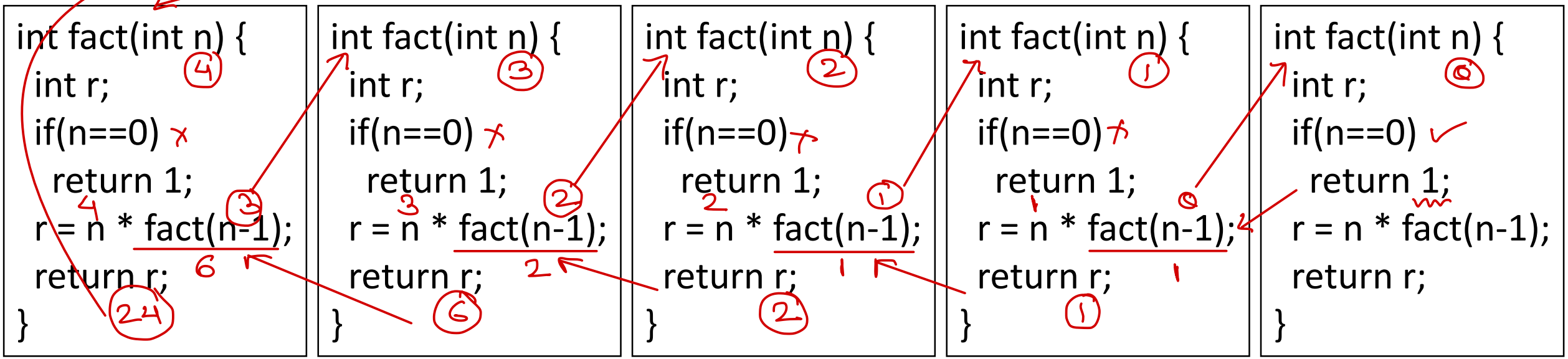
$2^2 = 2 * 2^1$

$2^1 = 2 * 2^0$

$2^0 = 1$

Recursion

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```



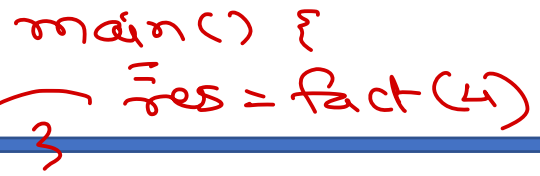
function activation record / stack frame is created on stack for each fn call.

* created when fn called
* destroyed when fn return

FAR cabins

1. local variables
2. arguments
3. return address
   - ↳ addr of next instru to be executed when fn returns.
   - = addr of next instru after fn call.

Recursion



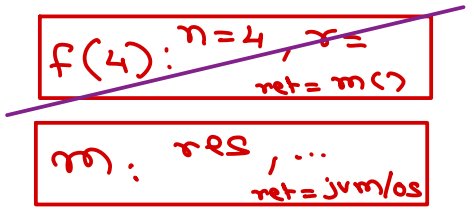

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

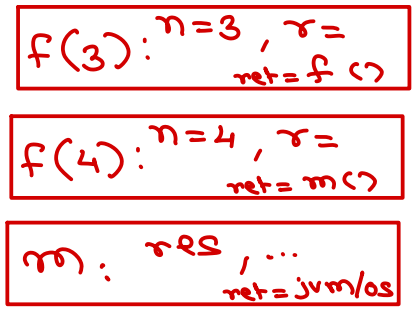
```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

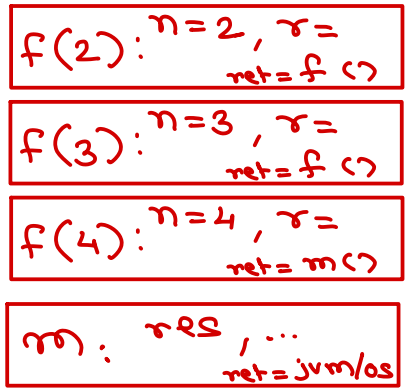
```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

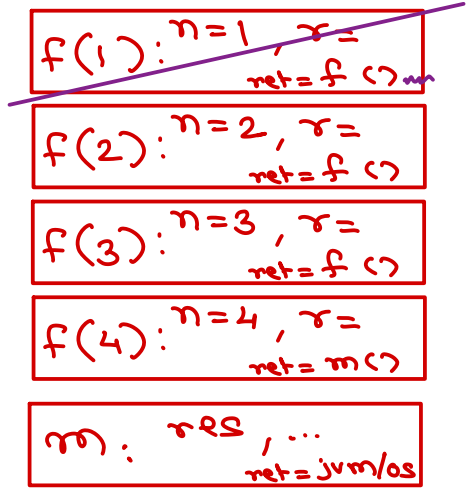
```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

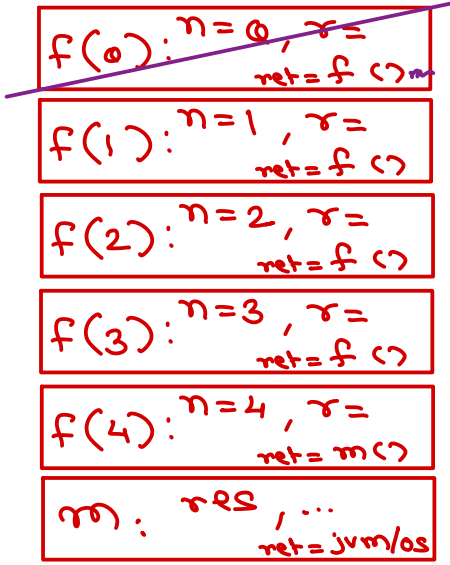
```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```



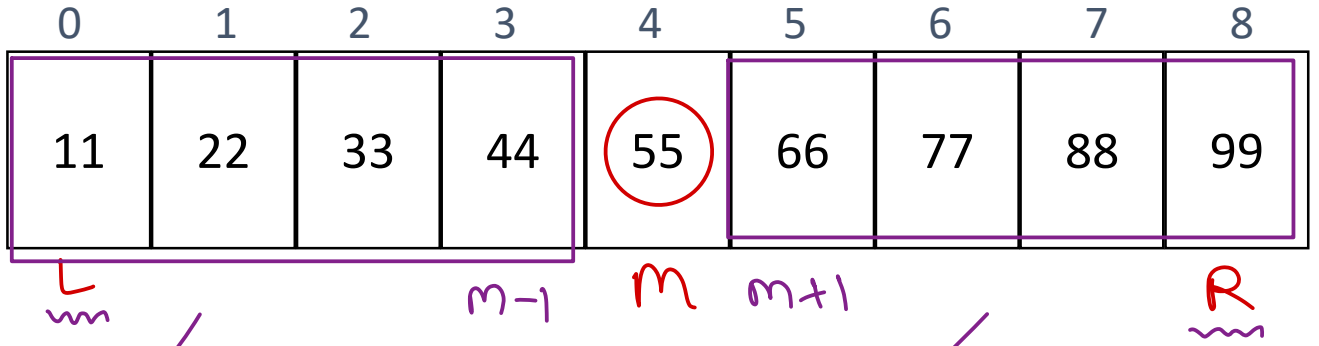








Binary Search




```
m=(l+r)/2;
if(key==a[m])
  return key
if( key < a[m])
    binSearch(l, m-1, key)
else
    binSearch(m+1, r, key)
```

base cond: right < left

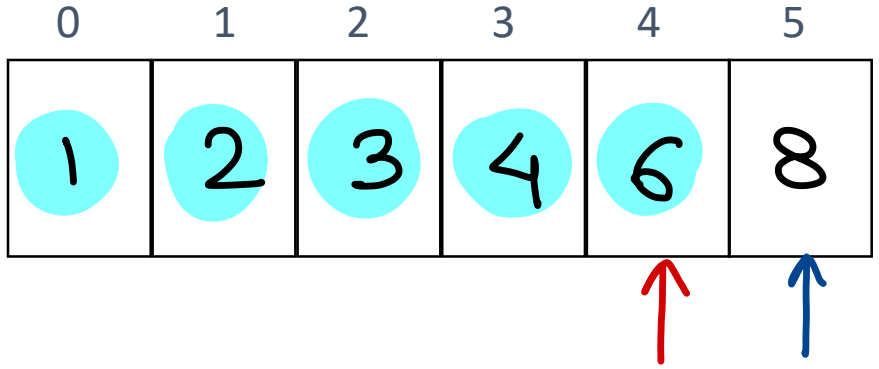
Sorting

- Arranging array elements in ascending or descending order.

- Popular sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Quick sort
  - Merge sort
  - Heap sort

6 4 2 8 3 1

```
for(i=0; i<5; i++) {
    for(j=i+1; j<6; j++) {
        if(a[i] > a[j])
            Swap(a[i], a[j]);
```



3

3

$i = (n-1) + (n-2) + (n-3) + \ldots + 1$

$i = n(n-1)/2$

$T \propto \frac{n^2-n}{2}$

$T \propto n^2 - n$

$T \propto n^2$

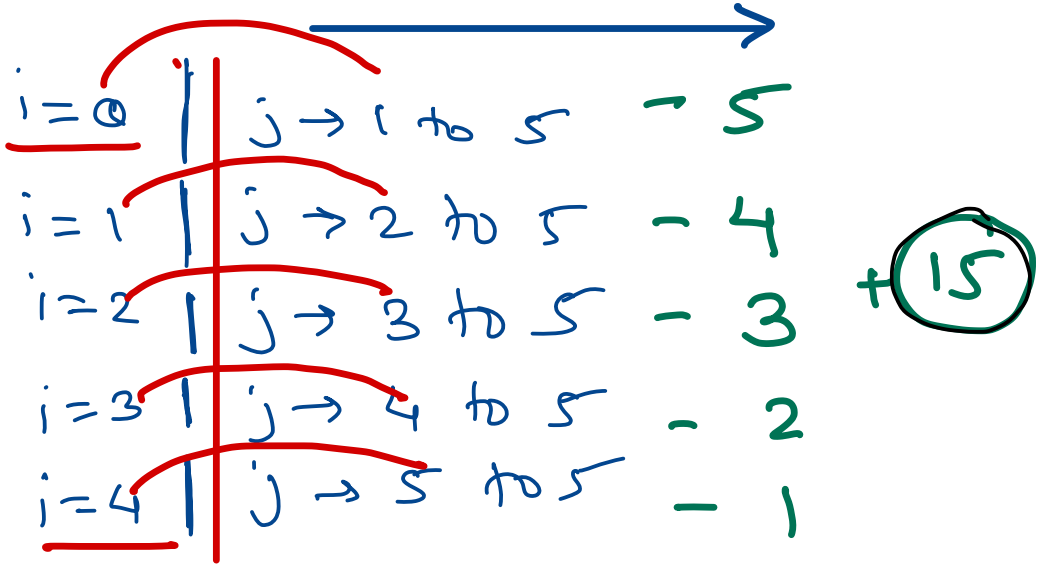
$O(n^2)$

Theory of Approximation

$n >> 1$

$n^2 >>>> n$

all lower order terms can be neglected.



Selection Sort

class Person : id, name, age.

0	1	2	3	4	5
1	5	6	4	7	3
B	X	P	G	N	Q
24	20	28	30	39	22

```
for(i=0; i<5; i++) {
    for(j=i+1; j<6; j++) {
        if(a[i].age > a[j].age)
            Swap(a[i], a[j]);
    }
}
```

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>

Data Structure & Algorithms

Sunbeam Infotech

Bubble Sort

6 4 2 8 3 1

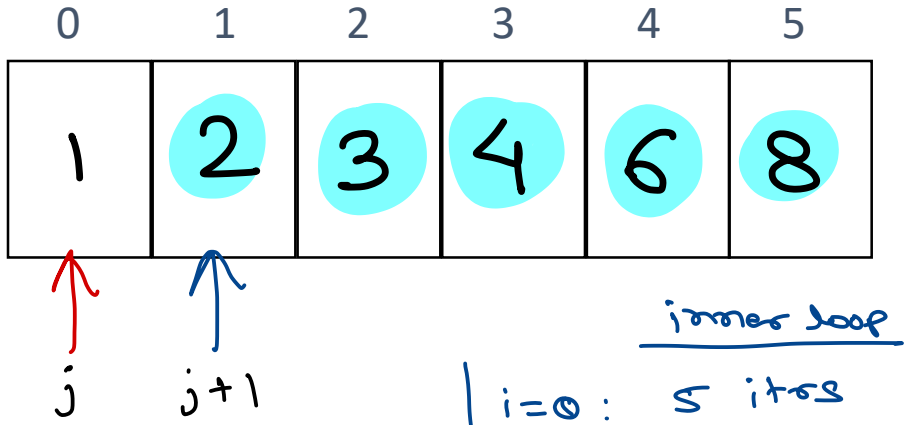
```
for(i=0; i<n-1; i++) {
    for(j=0; j<n-1; j++) {
        if(a[j] > a[j+1])
            swap(a[j], a[j+1]);
    }
}
```

$i = (n-1) * (n-1)$

$T \propto n^2 - 2n + 1$

$T \propto n^2$

$O(n^2)$



inner loop

$(n-1)$ Passes

i=0 :	5 itrs
i=1 :	5 itrs
i=2 :	5 itrs + 25
i=3 :	5 itrs
i=4 :	5 itrs

$(n-1)$ comp

Bubble Sort - improved Bubble Sort

```
for(i=0; i<n-1; i++) {
    for(j=0; j<n-1-i; j++) {
        if(a[j] > a[j+1])
            swap(a[j], a[j+1]);
    }
}
```

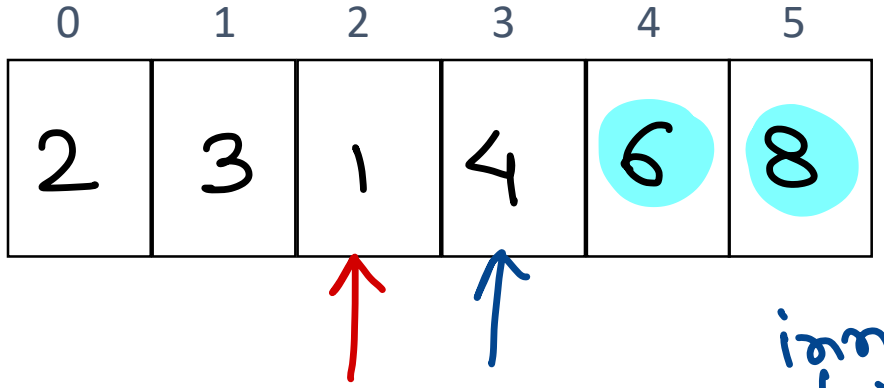
3

3

$i = (n-1) + (n-2) + \ldots + 1$

$T \propto n^2 - n$

$O(n^2)$



inner loop

i=0 : 5

i=1 : 4

i=2 : 3

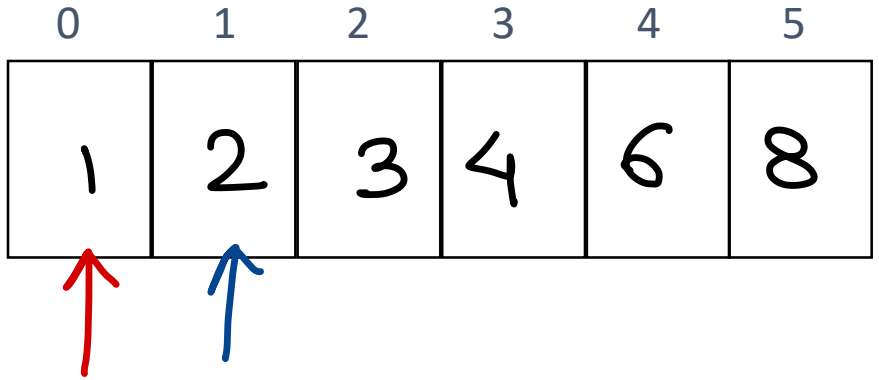
i=3 : 2

i=4 : 1

+15

Bubble Sort - more improved Bubble Sort

```
for(i=0; i<n-1; i++) {
    swapFlag = false;
    for(j=0; j<n-1-i; j++) {
        if(a[j] > a[j+1]) {
            swap(a[j], a[j+1]);
            swapFlag = true;
        }
    }
    if(swapFlag == false)
        break;
}
```



Best case : array already sorted.

* inner loop is executed only once.

$i = n-1$

$T \propto n-1$

$O(n)$

Insertion Sort

6 4 2 8 3 1

1 3 4 6 8 2

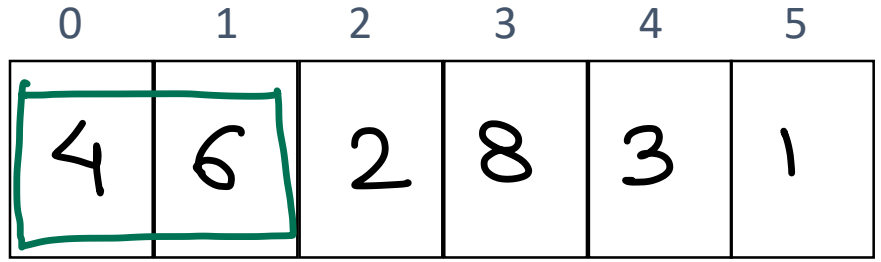
1 2 3 4 6 8

t = 2

-1 0 1 2 3 4 5

1 2 3 4 6 8

1



2 4 6

i=2

2 4 6 8

i=3

2 3 4 6 8

i=4

1 2 3 4 6 8

i=5

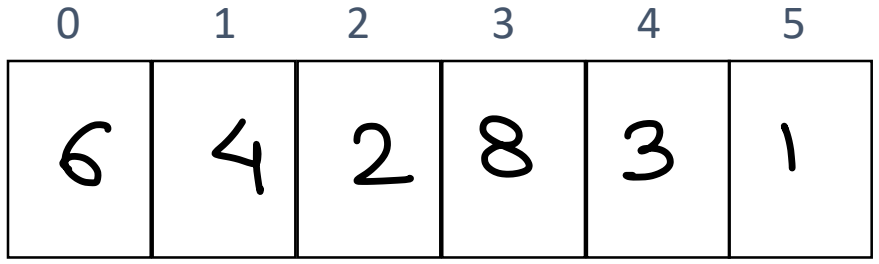
Insertion Sort

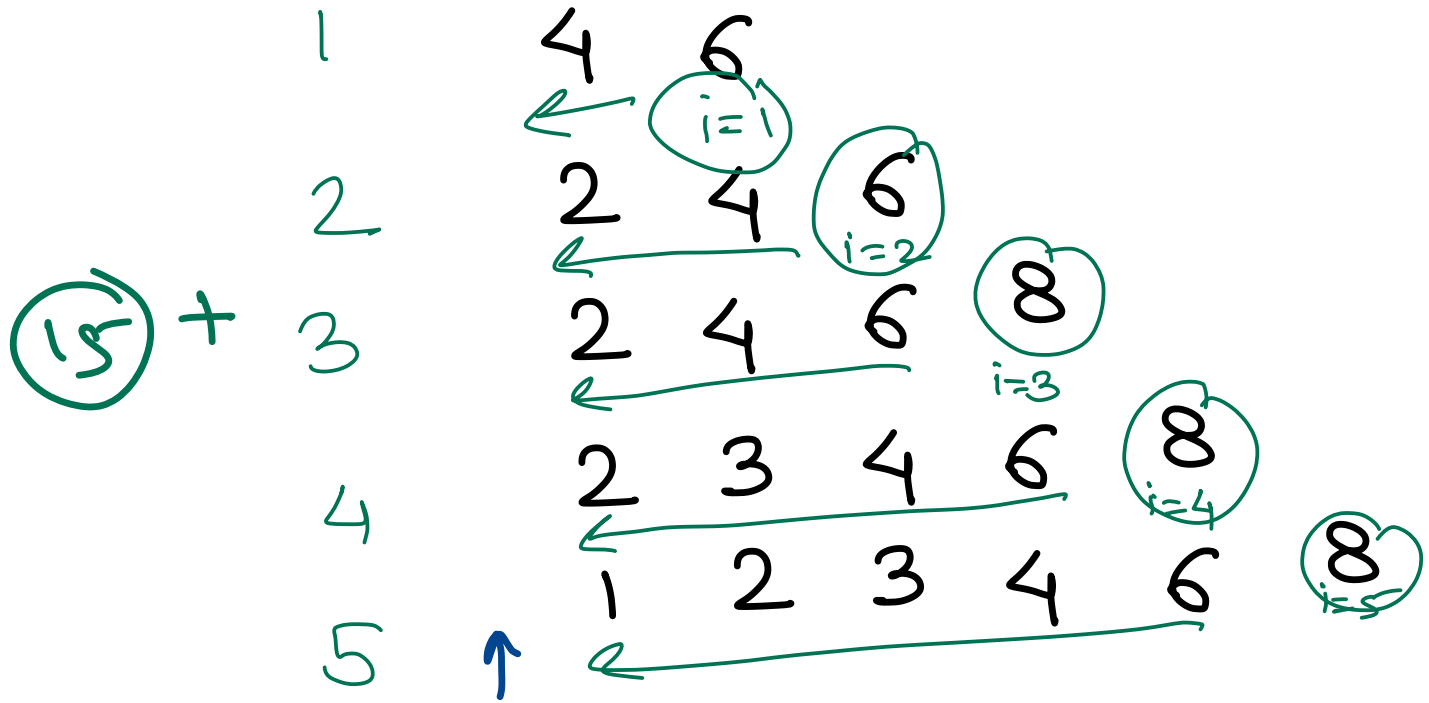
$i = 1 + 2 + \ldots + (n-1)$

$i = n(n-1)/2$

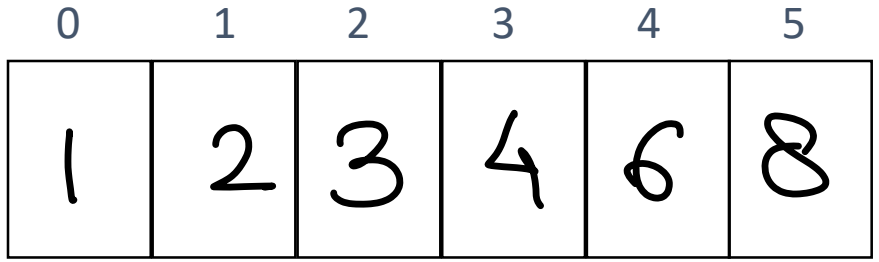
$T \propto n^2 - n$

$O(n^2)$





Insertion Sort



Best Case

$T \propto n$

$O(n)$

Linked List

- Linked List is list of items linked together.
- Each item in linked list is called as Node.
- Each node contains data and pointer/reference to the next node.
- Linked list is linear data structure.

→ can grow/shrink at runtime
→ sequential access only.
→ overheads
→ not contiguous alloc.

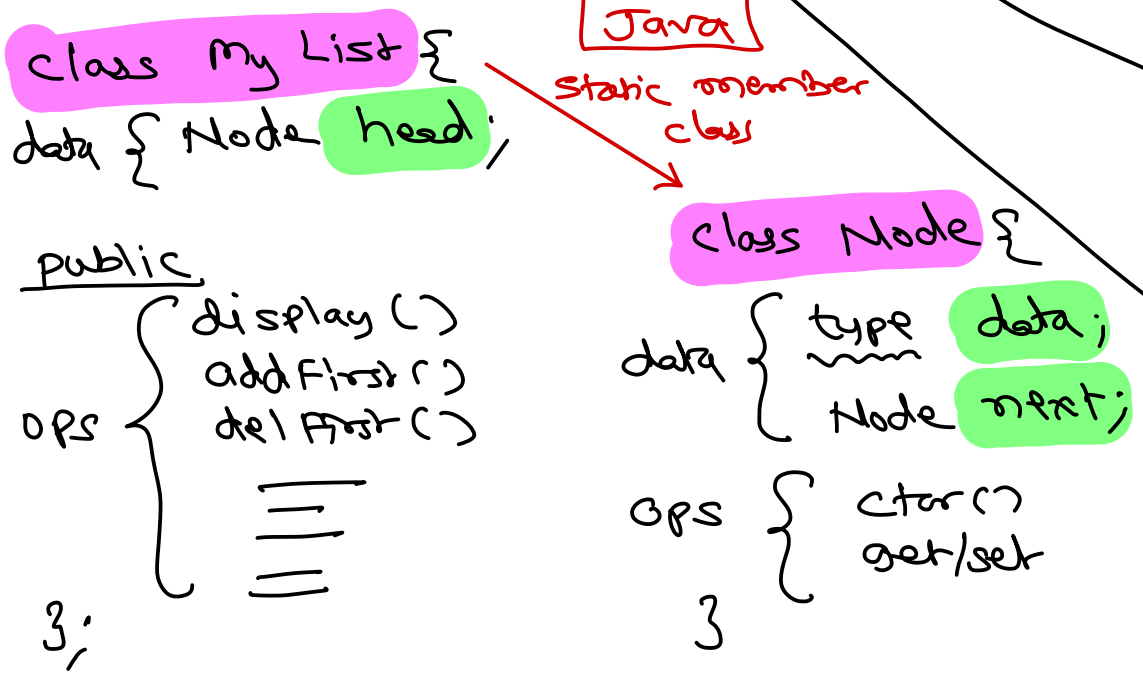
Arrays

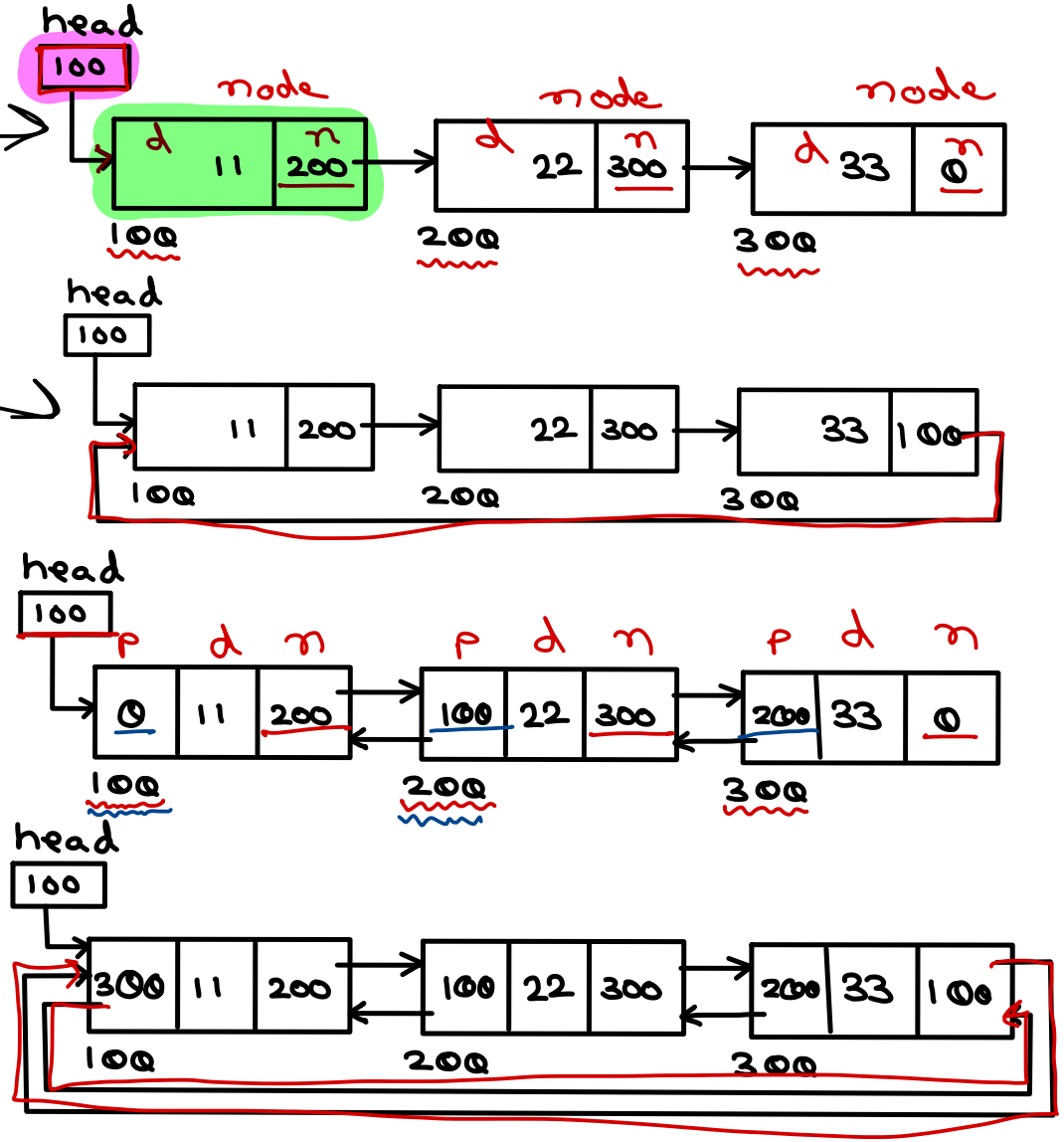
1. random access → O(1)
2. fixed size
   - cannot grow/shrink at runtime.
3. contiguous allocation
4. no overheads
   - no extra space required other than data elements.

- Linked list ADT
  - addFirst()
  - addLast()
  - addAtPos()
  - deleteFirst()
  - deleteLast()
  - deleteAsPos()
  - deleteAll()
  - traverse() or display()

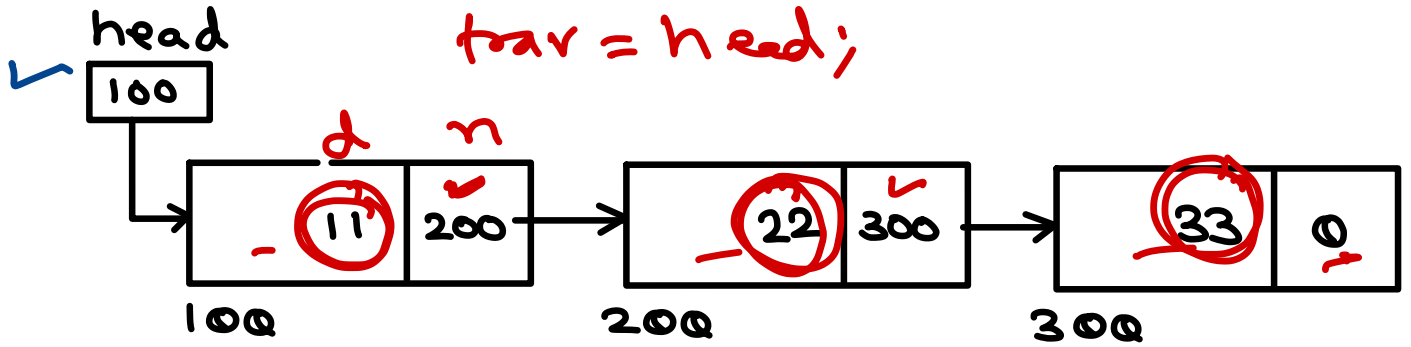
Linked List

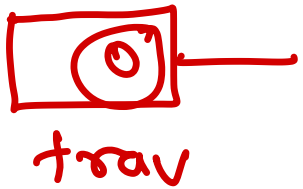
- There four types of linked list.
  - Singly linear linked list
  - Singly circular linked list
  - Doubly linear linked list
  - Doubly circular linked list





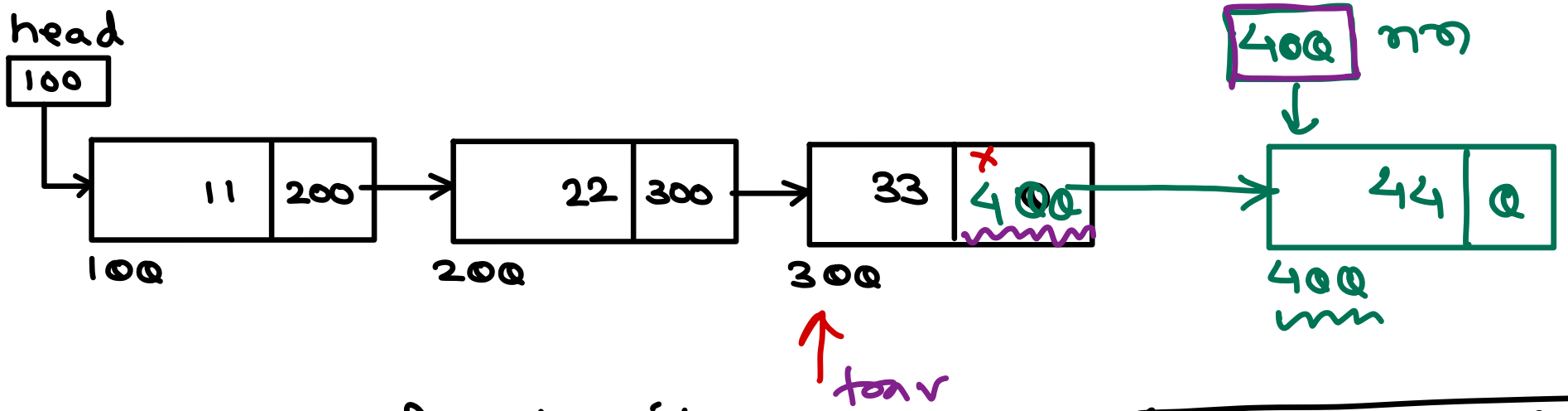
Linked List – display






```
while ( trav != null )
{
    Pf ( trav.data ) --
    trav = trav.next;
}
```

Linked List – addLast()



① Create new node & init it.

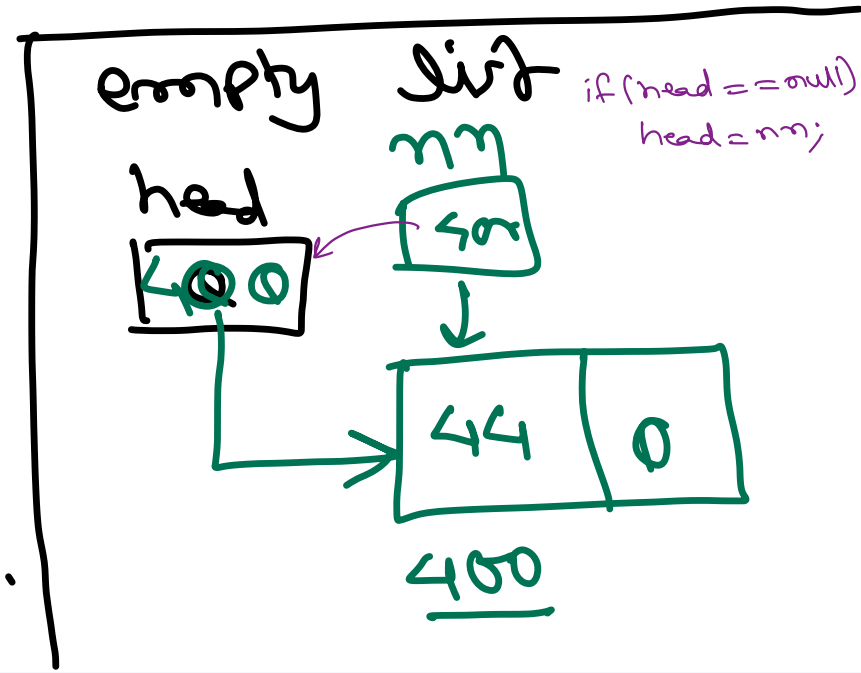
```
Node nn = new Node(val);
```

② traverse till last node.

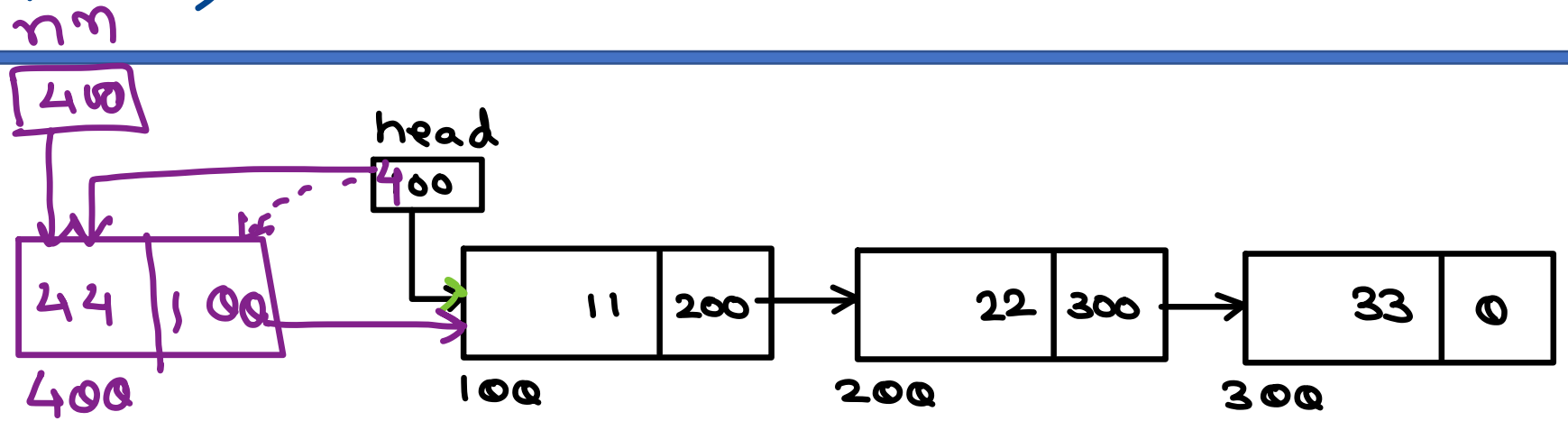
```
Node trav = head;
while(trav.next != null)
    trav = trav.next;
```

③ add new node into next of last node.

```
trav.next = nn;
```



Linked List - add First()



① alloc & init new node

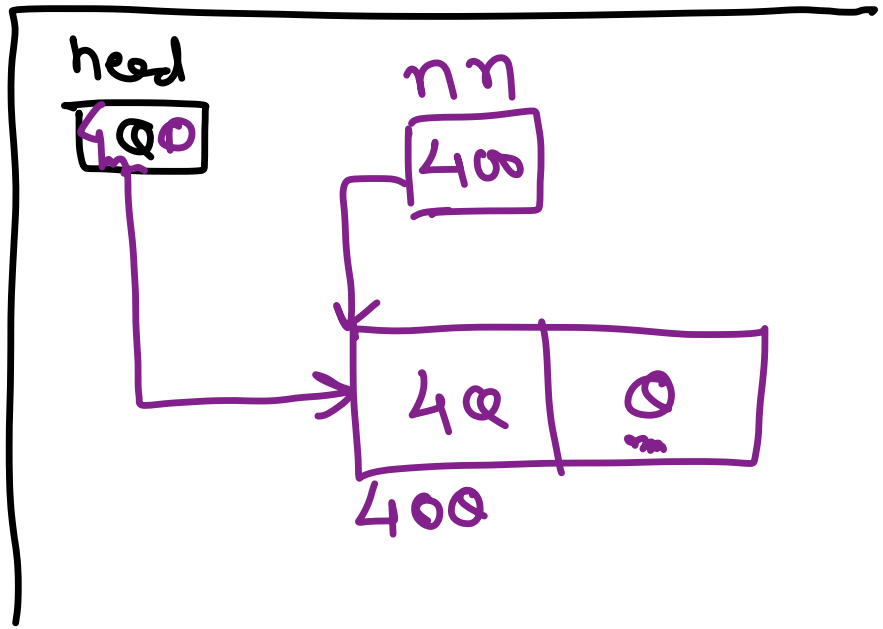
```
Node nn = new Node(val);
```

② new node next should first node addr

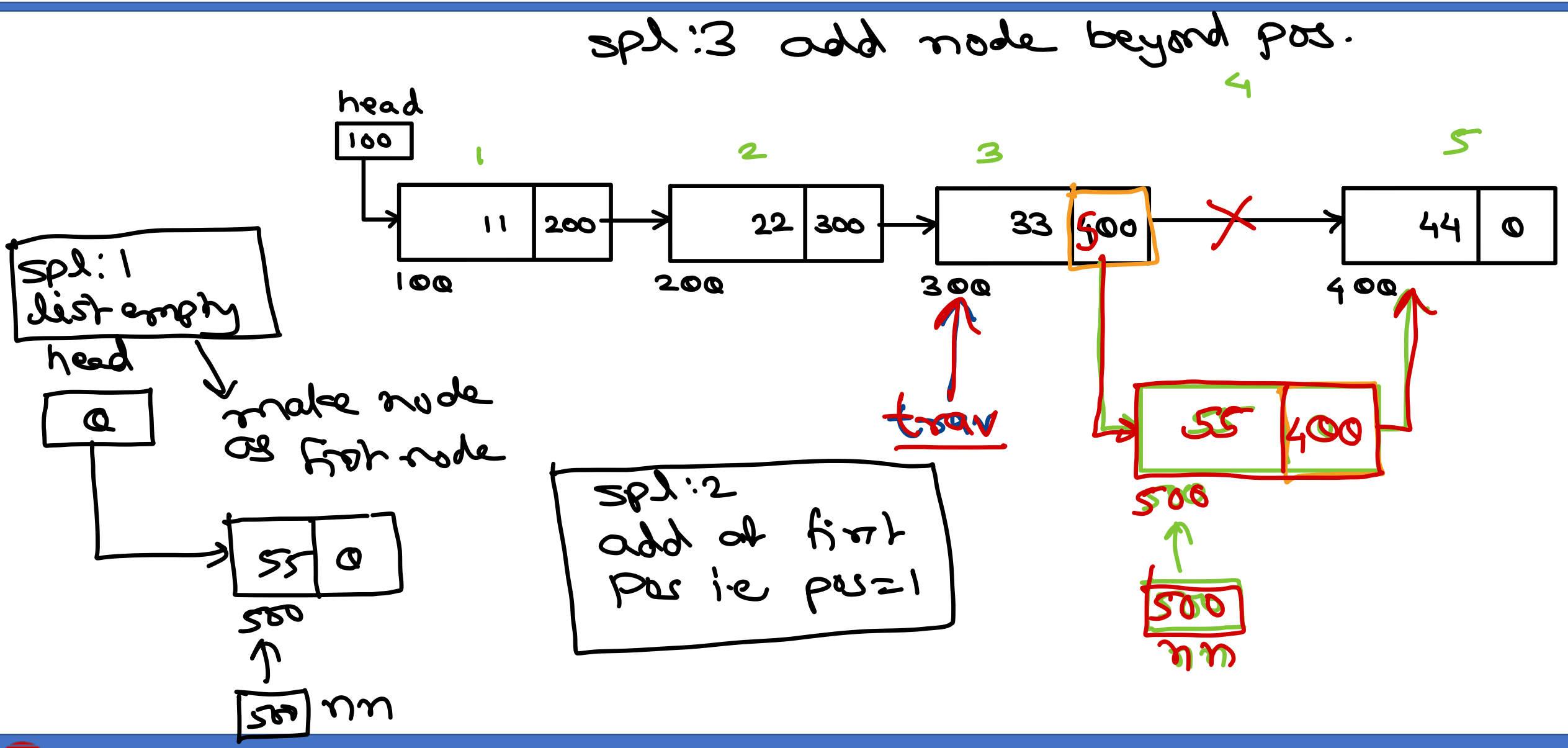
```
nn.next = head;
```

③ head should be new node.

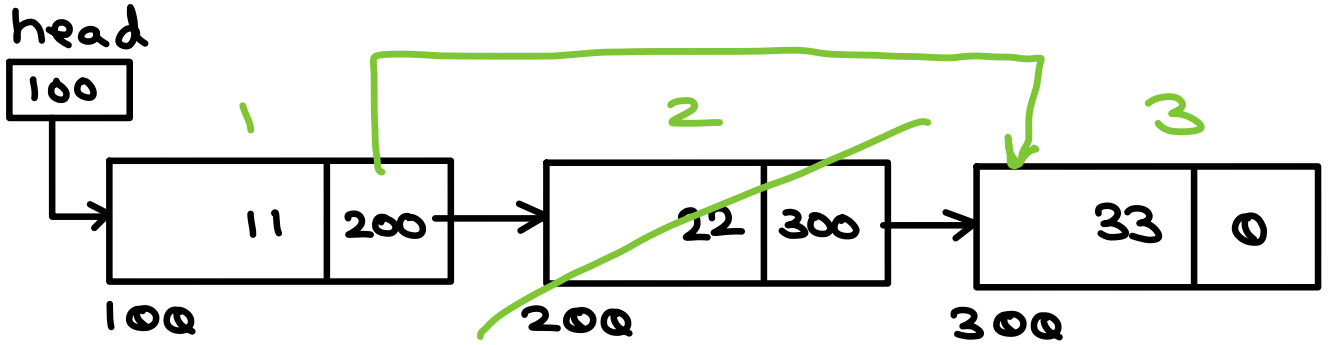
```
head = nn;
```

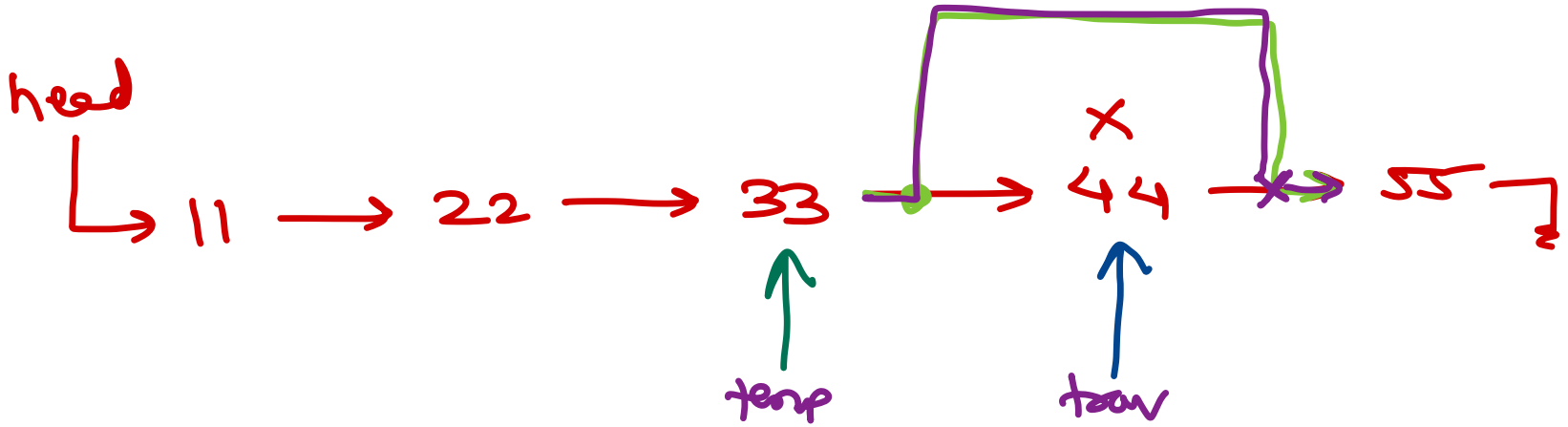


make before break



Linked List


head
100
1
2
3
11
200
22
300
33
0
100
200
300


head
11
22
33
x
44
55
temp
temp

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>

Data Structure & Algorithms

Sunbeam Infotech

Linked List - Singly List

1) display: O(n)

2) addLast: O(n)

3) addFirst: O(1)

4) addAtPos: O(pos)

   avg: O(n)

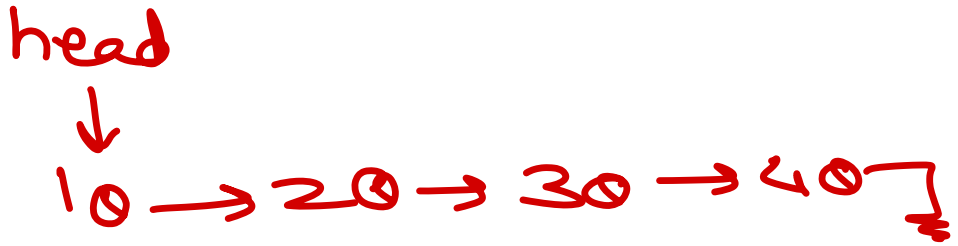
5) delFirst: O(1)

6) delLast: O(n)

7) delAtPos: O(pos)

   avg: O(n)

8) delAll: O(1) <- Java. (GC)



Linked List - Singly List

1) display : O(n)

2) addLast : O(1)

3) addFirst : O(1)

4) addAtPos : O(pos)

avg : O(n)

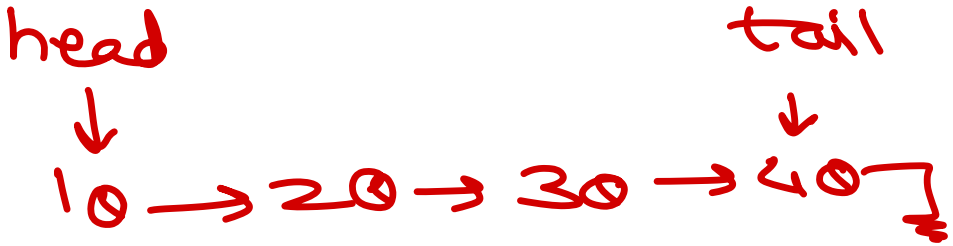
5) delFirst : O(1)

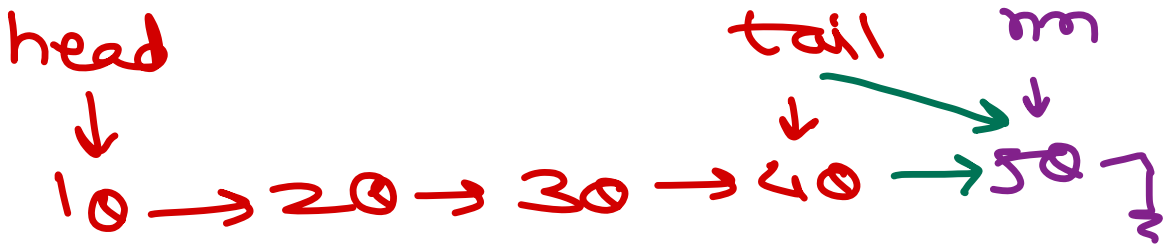
6) delLast : O(n)

7) delAtPos : O(pos)

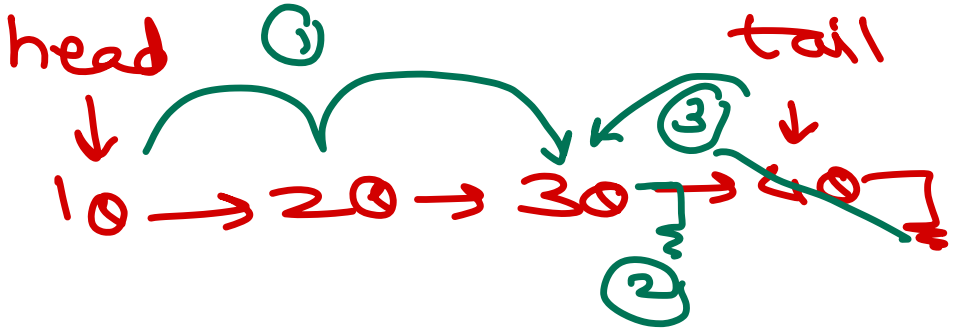
avg : O(n)

8) delAll : O(1) ← Java. (GC)



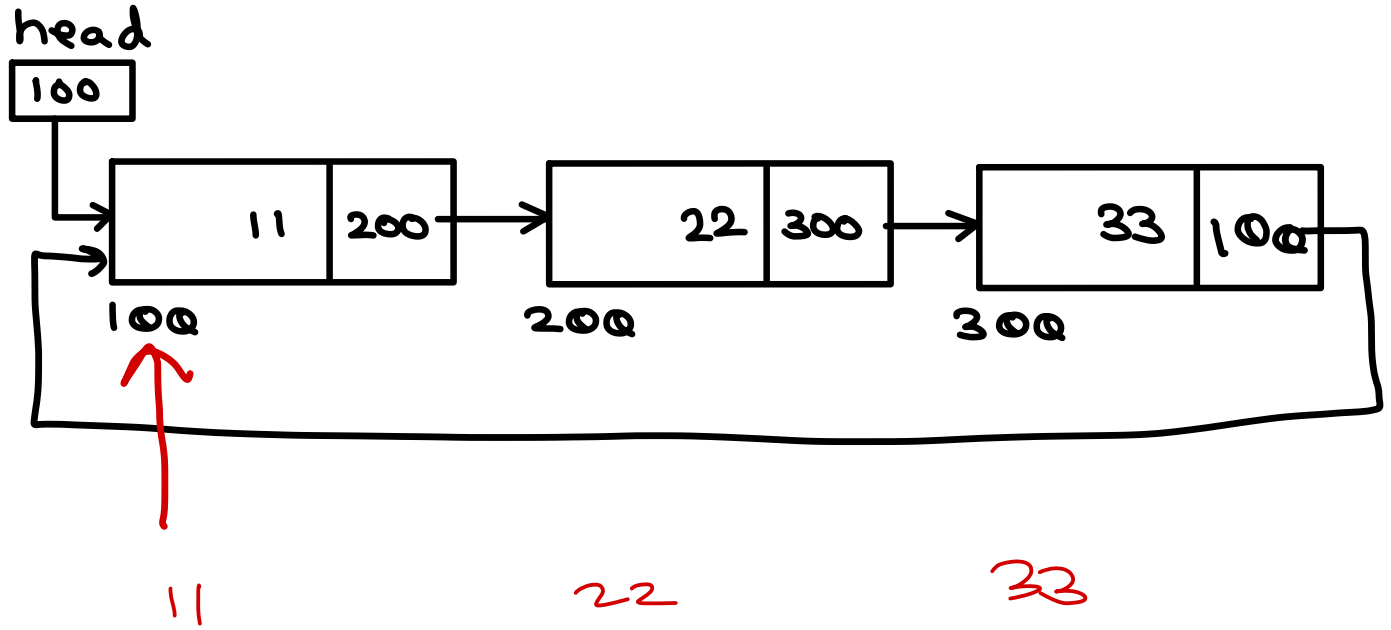



```
tail.next = nn;
tail = nn;
```



Linked List – Singly Circular List.

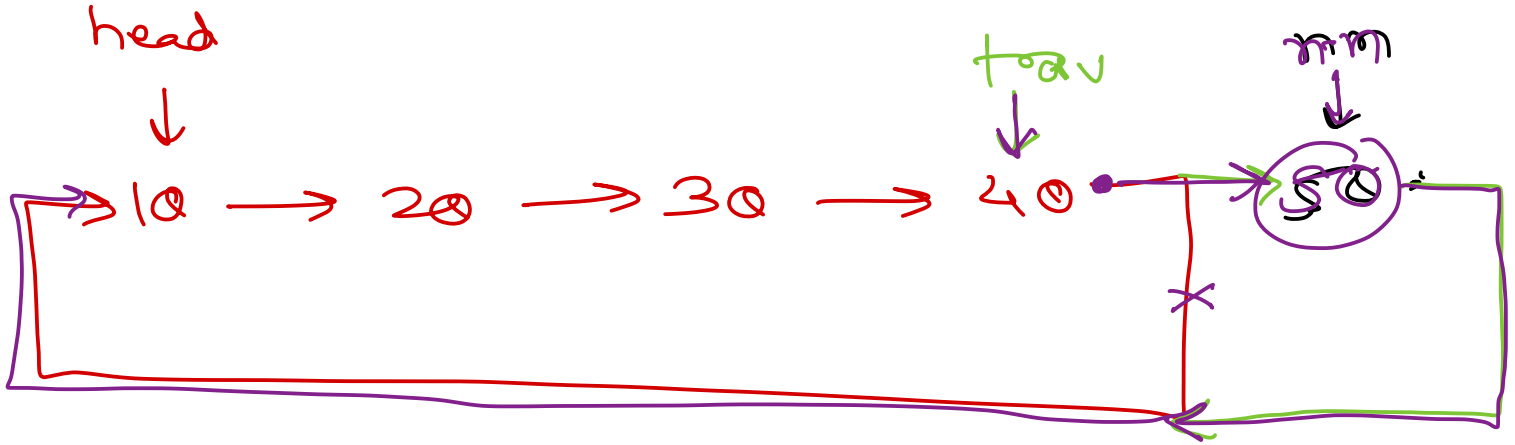
```
trav = head;
do {
    pf(trav.data);
    trav = trav.next;
} while(trav != head)
```



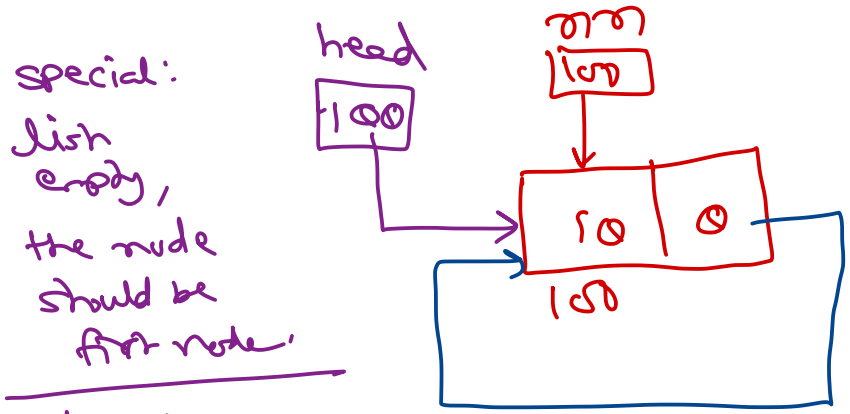
head
0

if list is empty, return;

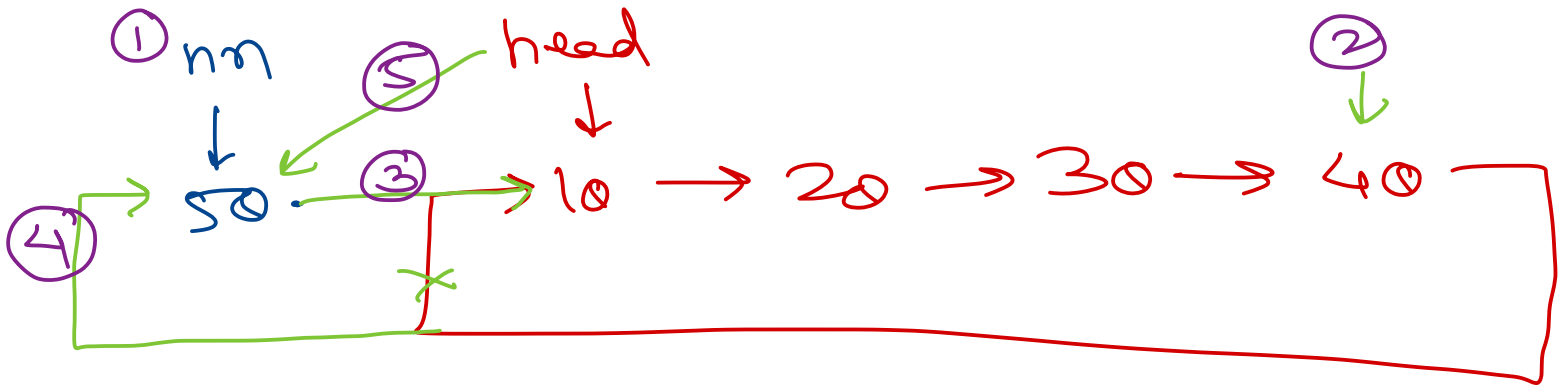
Linked List — addLast()



1. Create nn & init
2. traverse till last node (trav)
3. nn.next = head
4. trav.next = nn

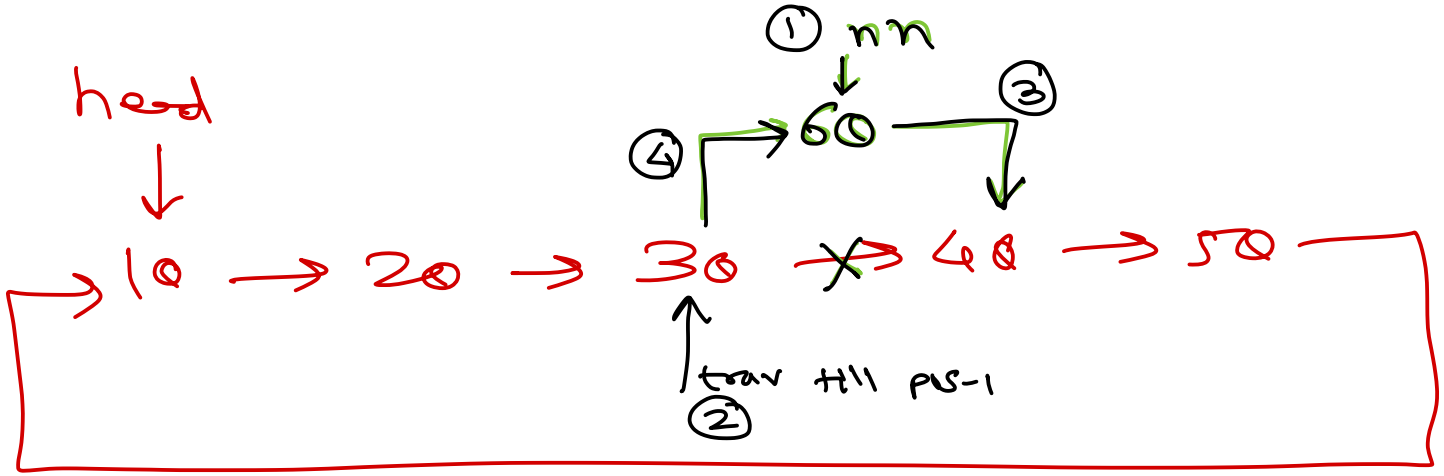


Linked List - addFirst()

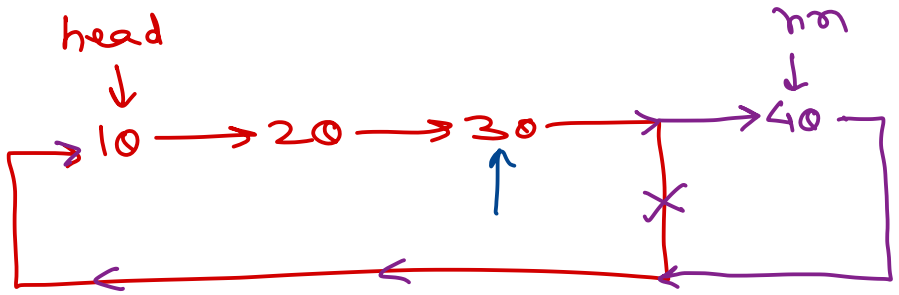


1. create nn & init
2. trav till last node
3. nn next = head
4. last node next = nn
5. head = nn;

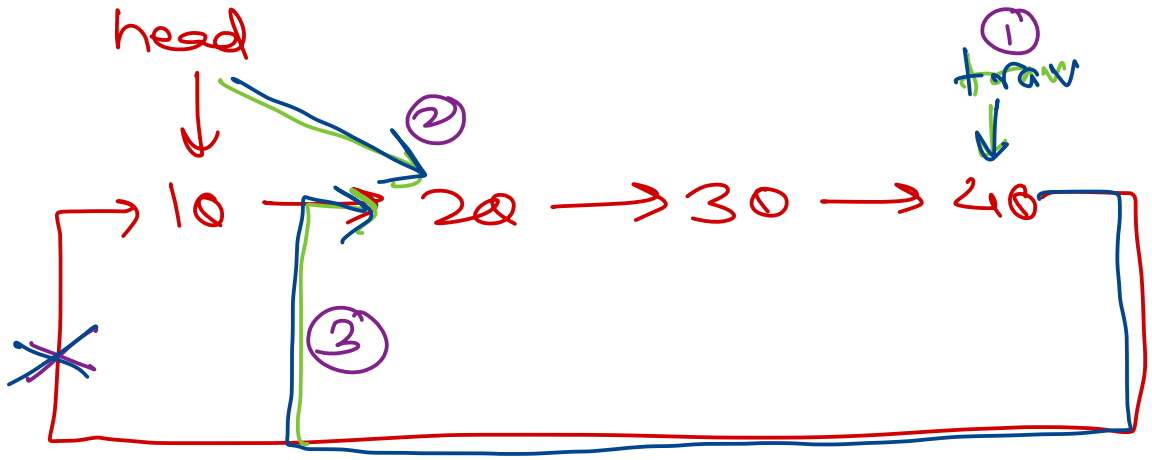
Linked List → add At Pos.



i=3 pos=8



Linked List → delFirst()

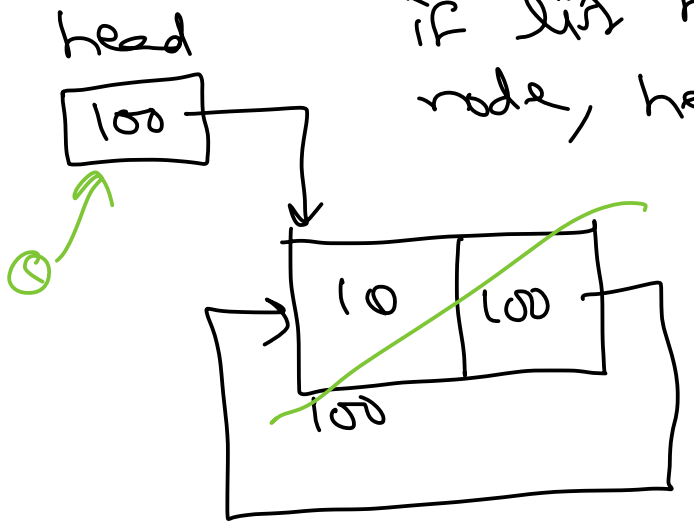


① trav till last node.

② take head to next node (2nd)

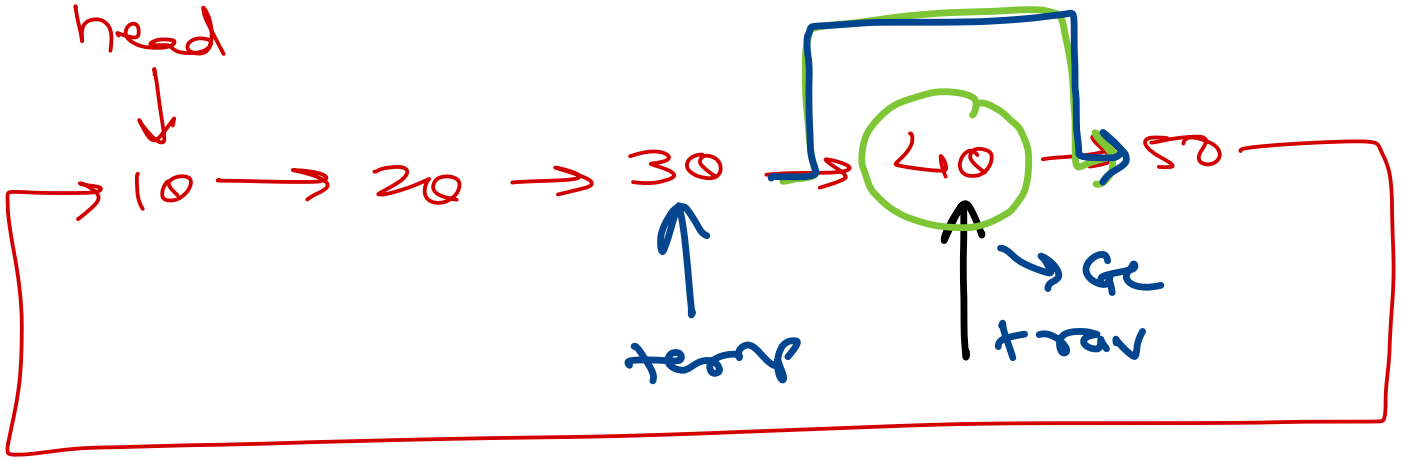
③ trav's next to new head.

if list is empty, throw exception.

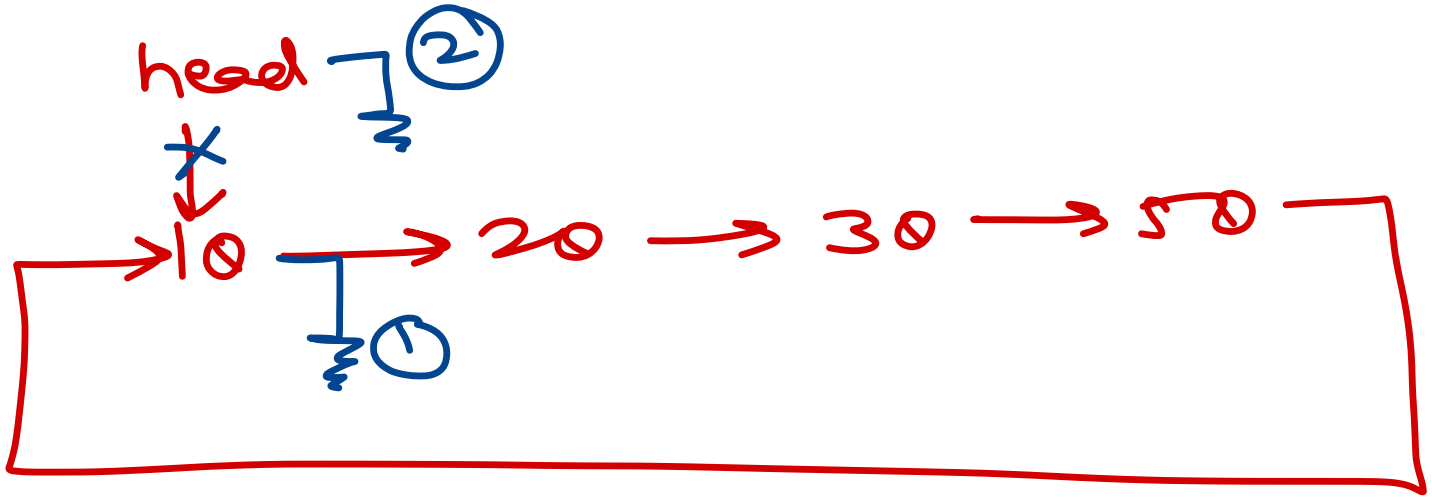


if list has single node, head = null;

Linked List → delAtPos()



Linked List



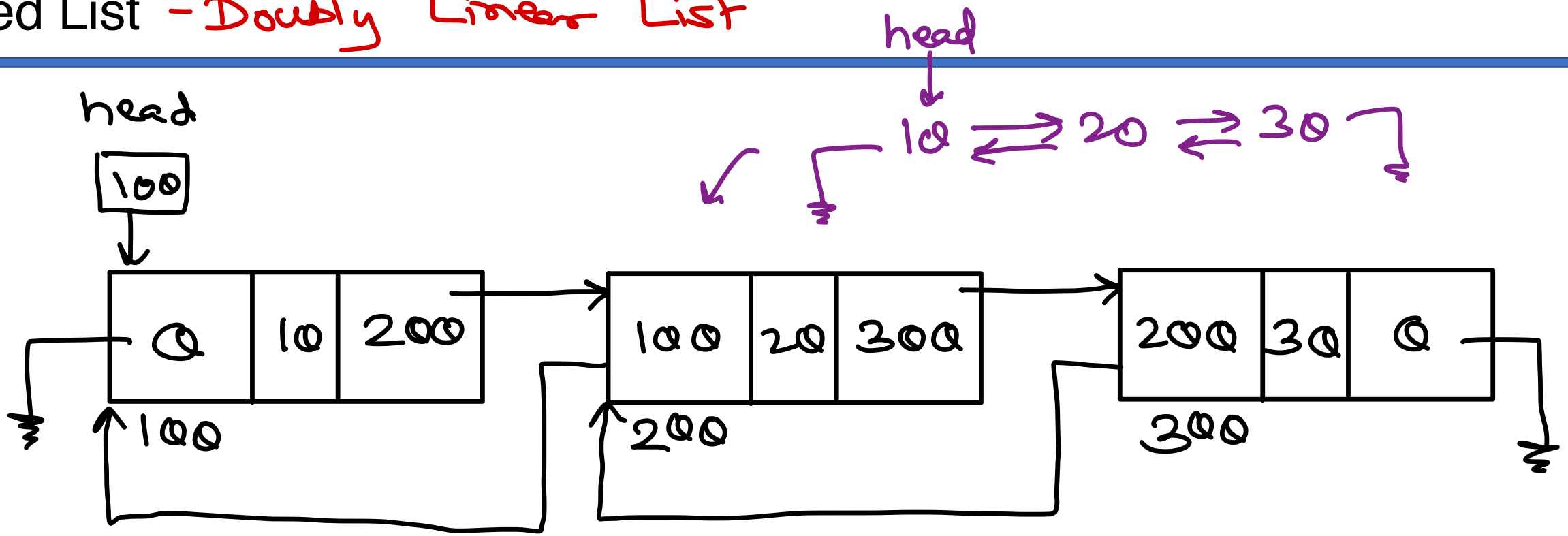
① Convert cir list into singly lin list.

```
head.next = null;
```

② make head null

```
head = null;
```

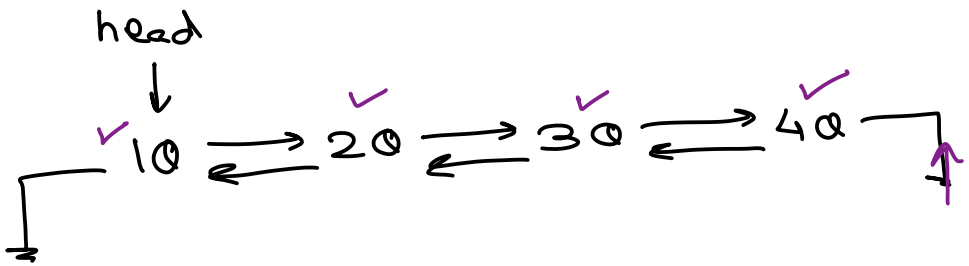
Linked List - Doubly Linear List



next → addr of next node

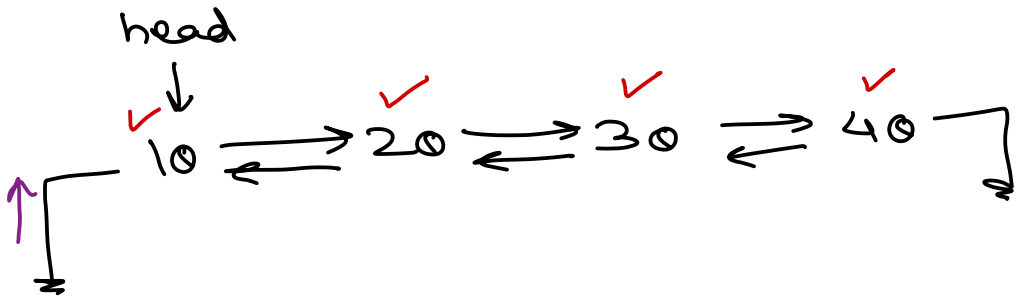
prev → addr of prev node.

Linked List – display()




```
trav = head;
while(trav != null)
{
    pf(trav.data);
    trav = trav.next;
}
```

forward display



① traverse till last node.

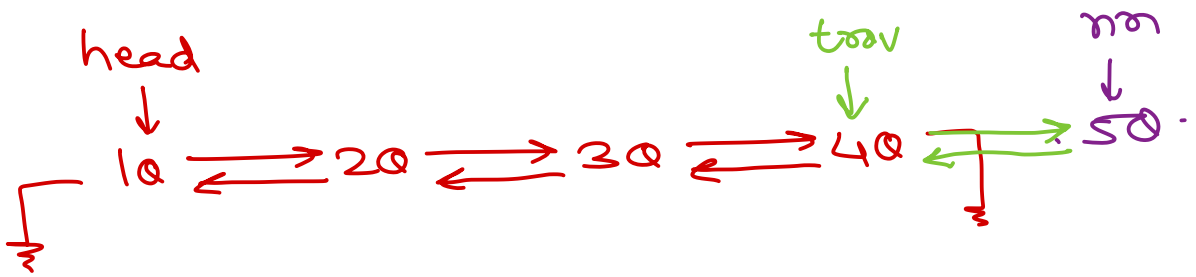
```
trav = head;
while(trav.next != null)
    trav = trav.next;
```

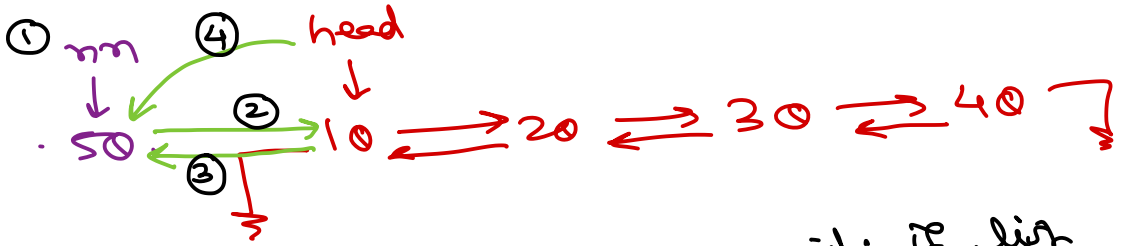
② trav & print each node in rev dir.

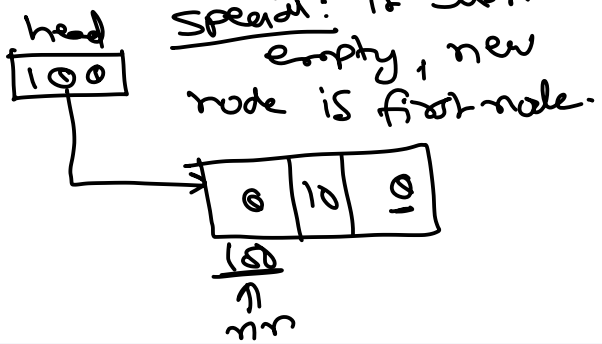
```
while(trav != null)
{   print(trav.data)
    trav = trav.prev;
}
```

reverse display

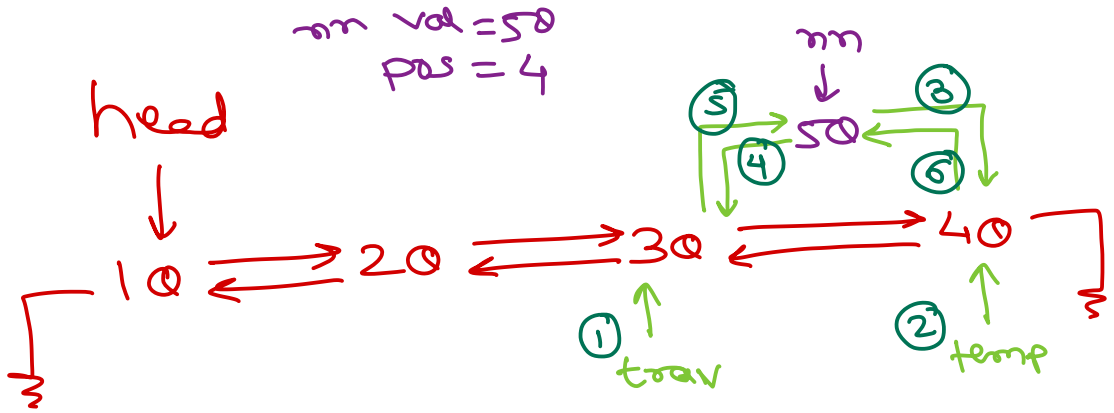
Linked List – addLast() /addFirst()

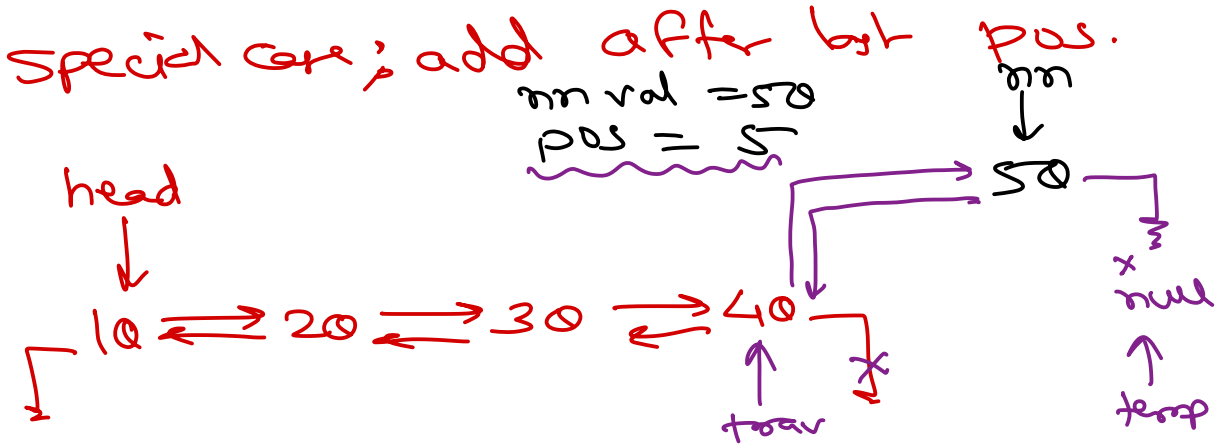




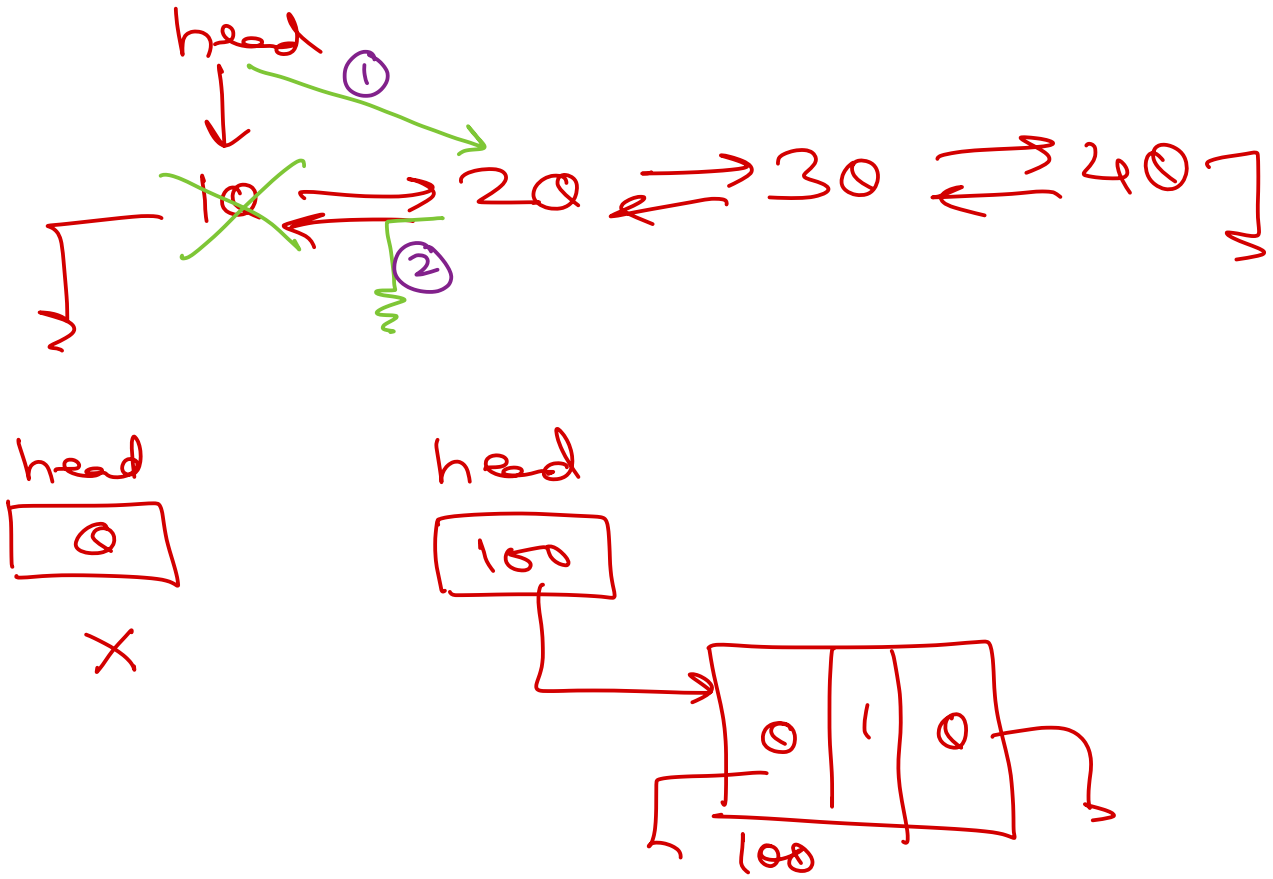


Linked List

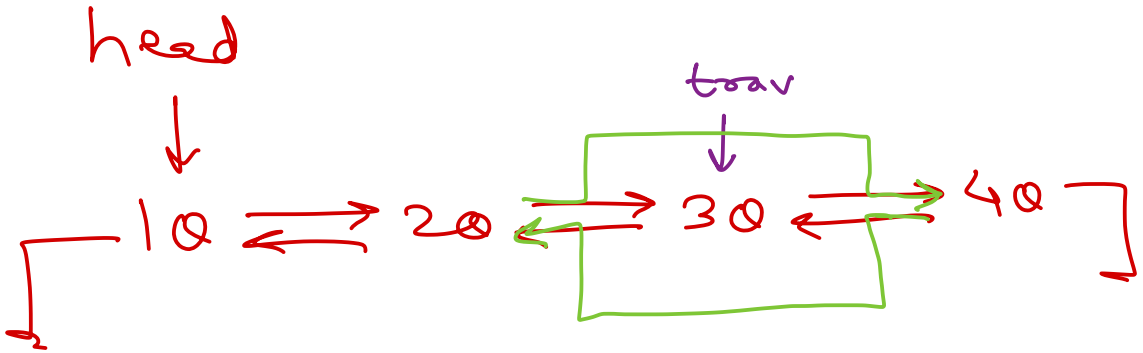

nn val = 50
pos = 4
head
nn
10
20
30
40
50
1
2
3
4
5
6
trav
temp


special care; add after last pos.
nn val = 50
pos = 5
head
nn
10
20
30
40
50
trav
temp
null

Linked List – del first()



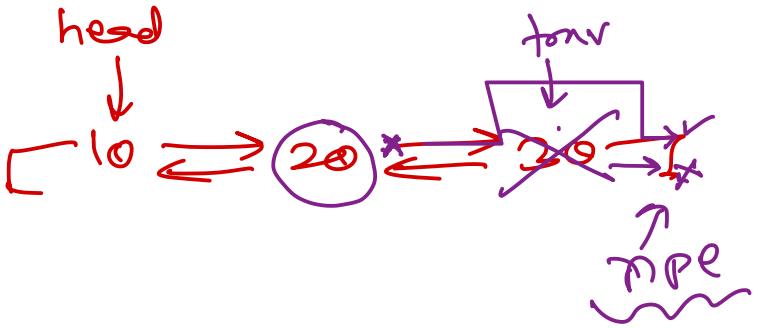
Linked List – del 'last'



① traverse till pos.

② previous node's next to trav's next;

③ next node's prev to trav's prev node



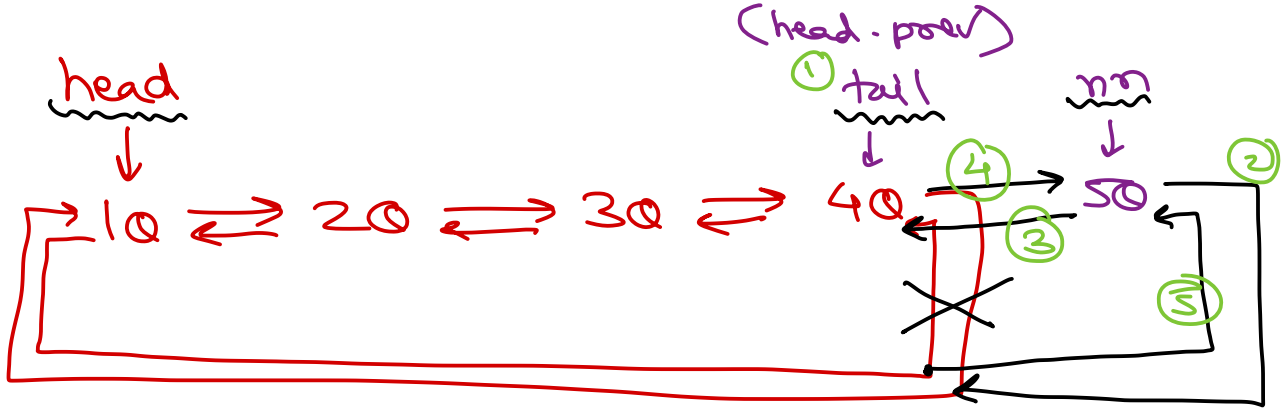
Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>

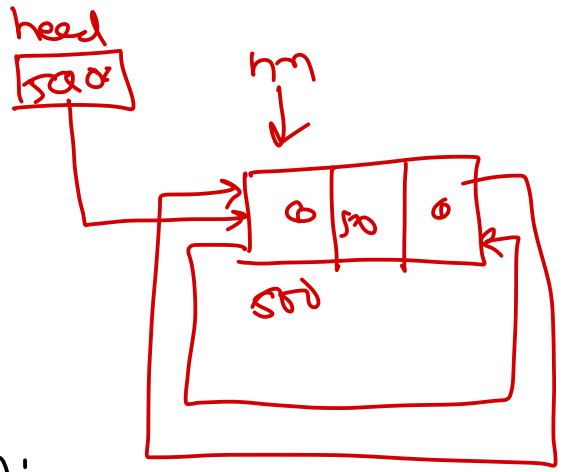
Data Structure & Algorithms

Sunbeam Infotech

Linked List - Doubly Circular List - addLast

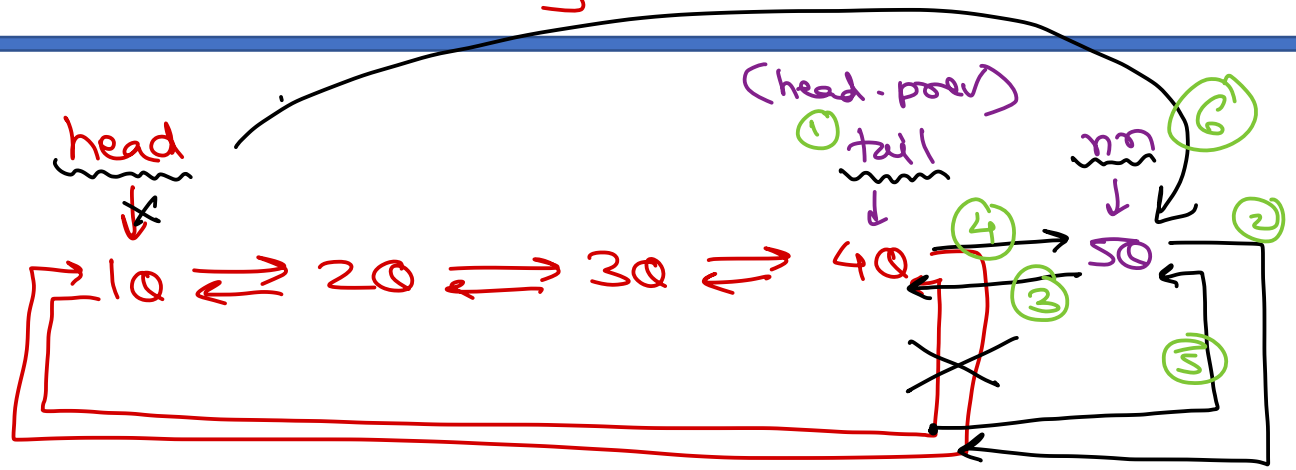


1. tail = head.prev;
2. nn.next = head;
3. nn.prev = tail;
4. tail.next = nn;
5. head.prev = nn;

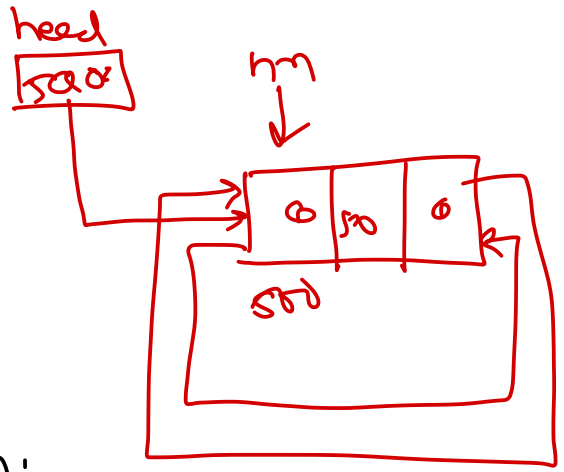


head = nn;
nn.next = head;
nn.prev = head;

Linked List - Doubly Circular List - add First

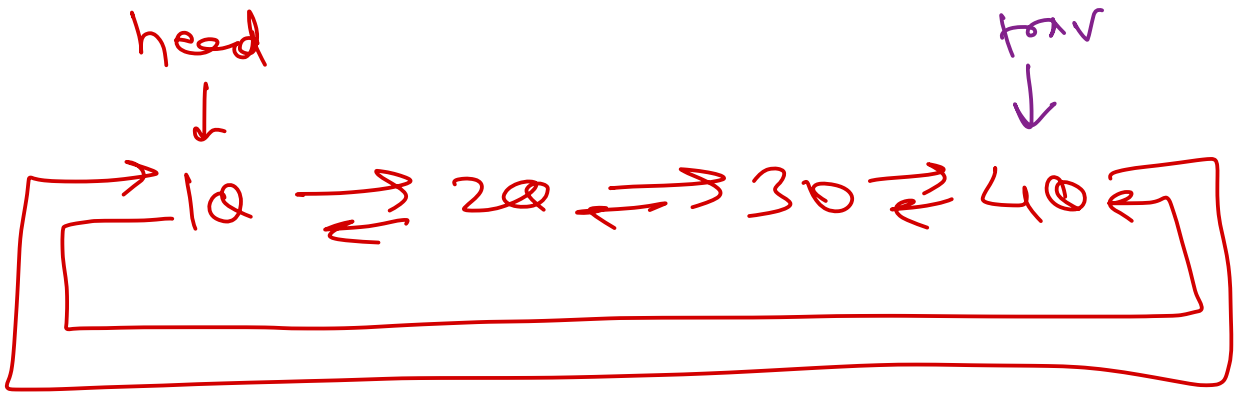


1. tail = head.prev;
2. nn.next = head;
3. nn.prev = tail;
4. tail.next = nn;
5. head.prev = nn;
6. head = nn;



head = nn;
nn.next = head;
nn.prev = head;

Linked List - Doubly Circular List



fwd display

```
trav = head;
do {
    pf(trav.data);
    trav = trav.next;
} while(trav != head);
```

rev display

```
trav = head.prev;
do {
    pf(trav.data);
    trav = trav.prev;
} while(trav != head.prev);
```

Linked List

Doubly Cir List

Time Complexities

1. add first - O(1)
2. add last - O(1)
3. del first - O(1)
4. del last - O(1)
5. display/trav - O(n)
6. add/del at pos - O(pos) ... O(n)
7. find ele - O(n)
   linear search

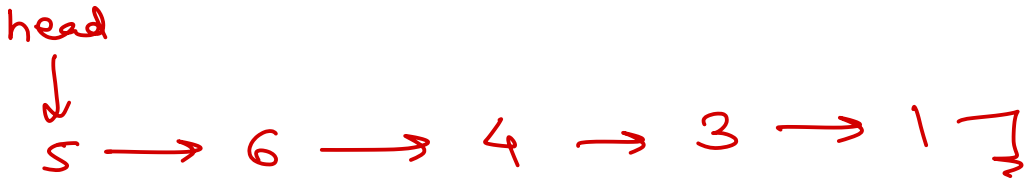
Linux Kernel

- all lists are doubly circular list.

1. job queue / process table
2. ready queue
3. waiting queue
4. message queue
5. inode table/ inode cache

Linked List – Competitive programming

- Sort the singly linked list.

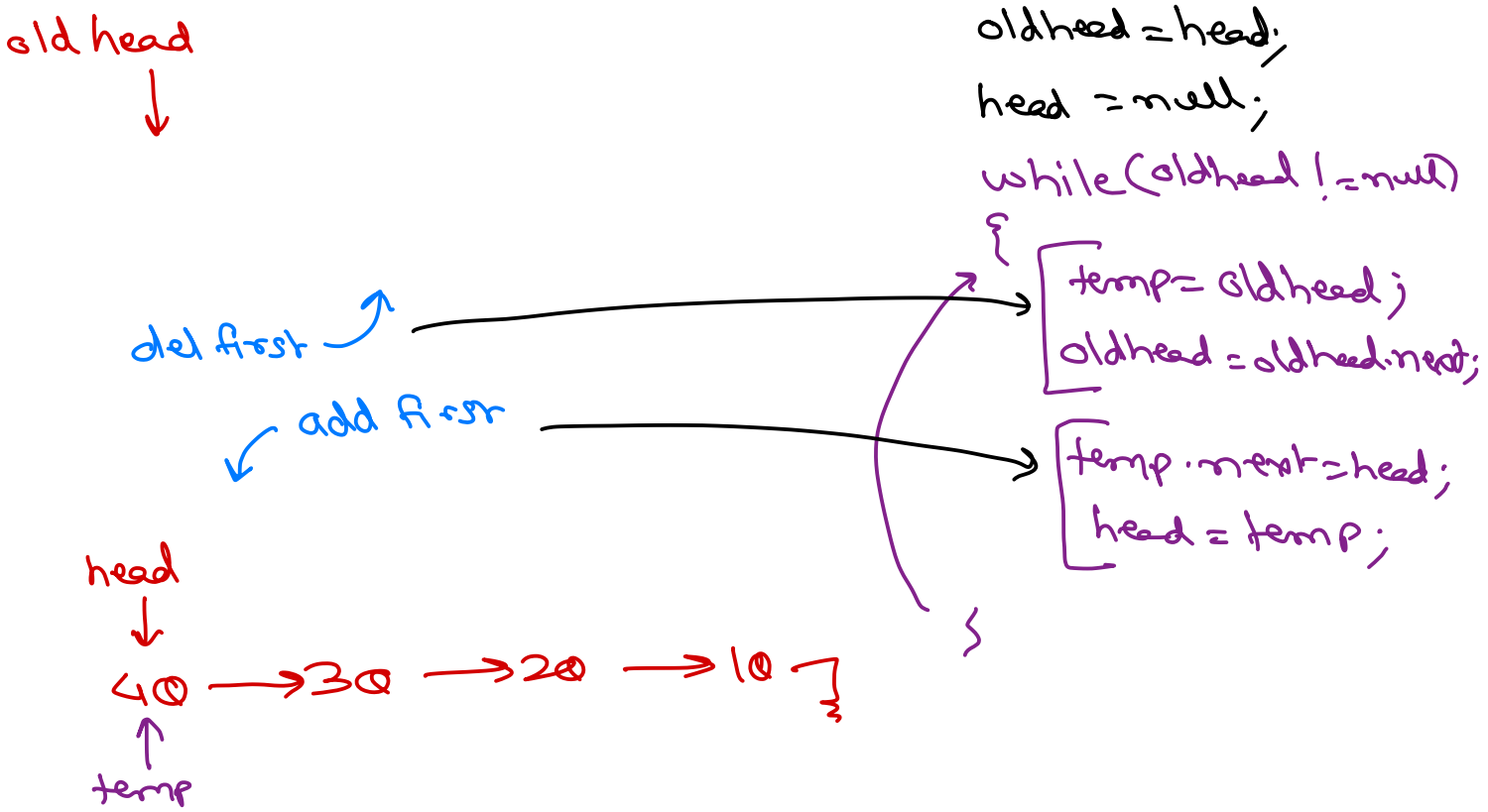


Selection sort

```
Node i,j; i.next!=null
for(i=head ; i !=null ; i=i.next) {
    for(j=i.next ; j !=null ; j=j.next) {
        if( i.data > j.data )
            swap(i.data, j.data);
    }
}
```

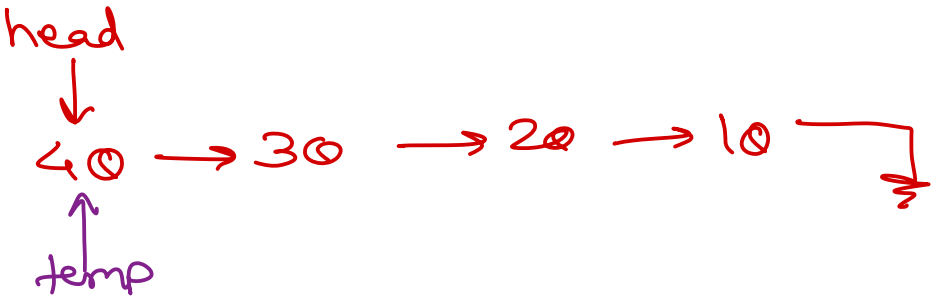
Linked List – Competitive programming

- Reverse singly linked list.



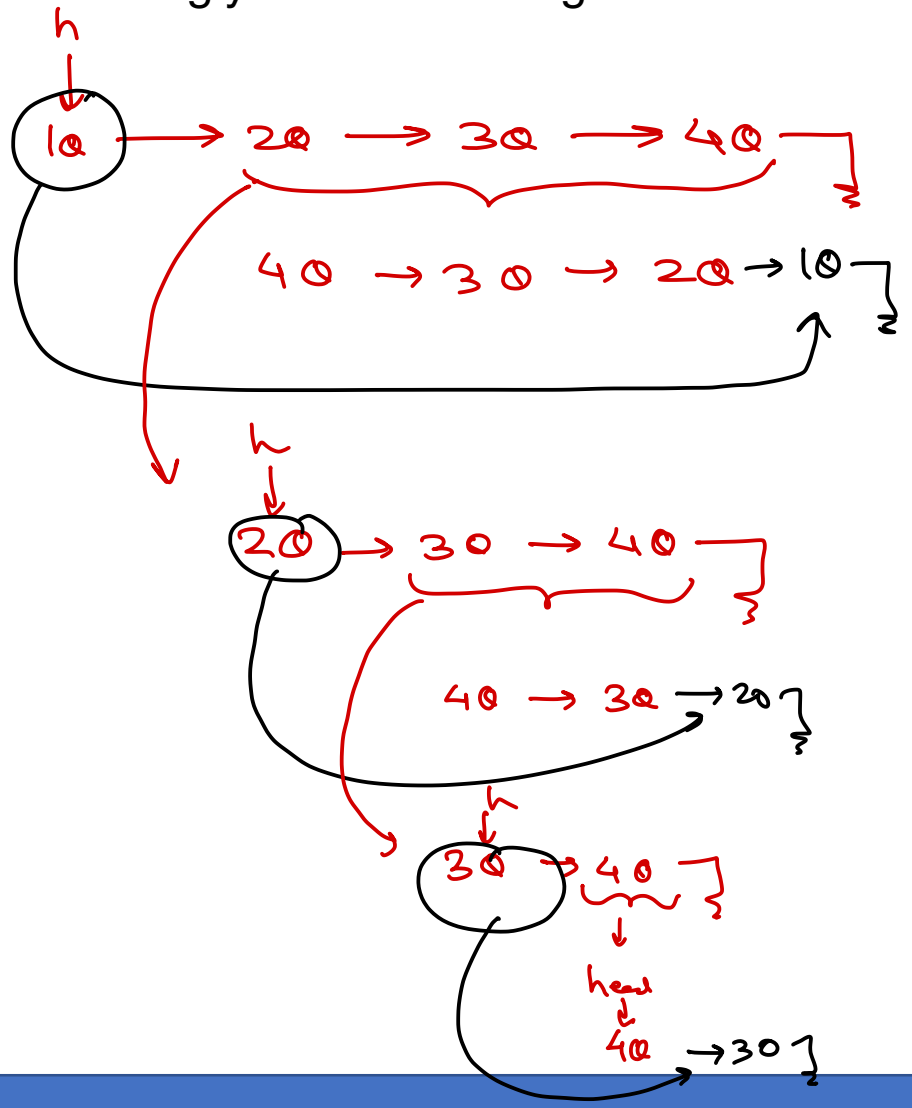
Linked List – Competitive programming

- Reverse singly linked list.



Linked List – Competitive programming

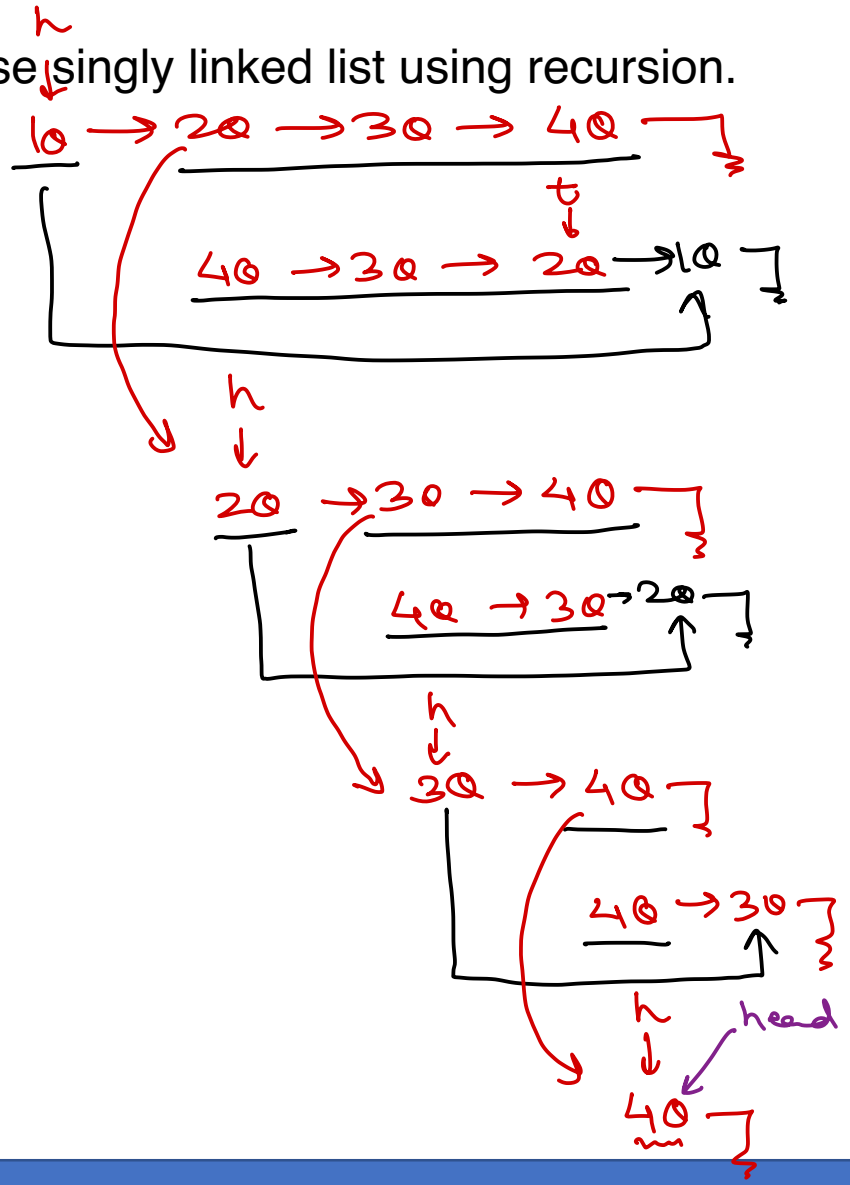
- Reverse singly linked list using recursion.




```
Node recRev(Node h) {
    if(h.next == null) {
        head = h;
        return h;
    }
    t = recRev(h.next);
    t.next = h;
    h.next = null;
    return h;
}
```

Linked List – Competitive programming

- Reverse singly linked list using recursion.



① if list has single node, mark it as head & return it (it is last node)

① reverse rest of list (from next)

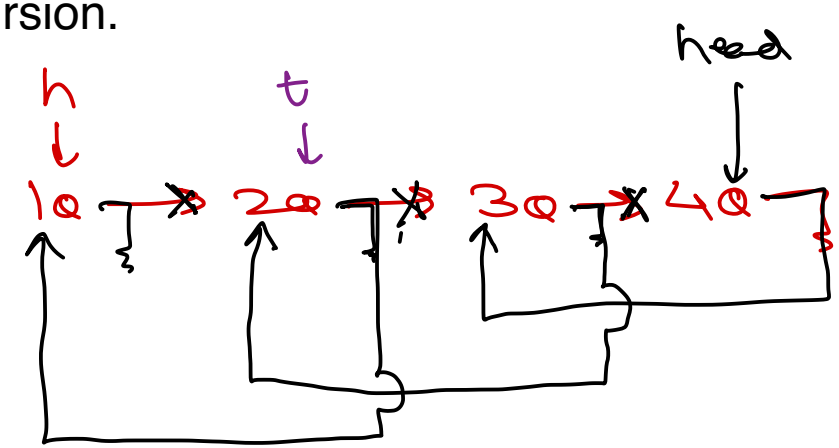
② add cur node (h) after last node (t)

③ make cur node next as null.
(now cur node became last node).

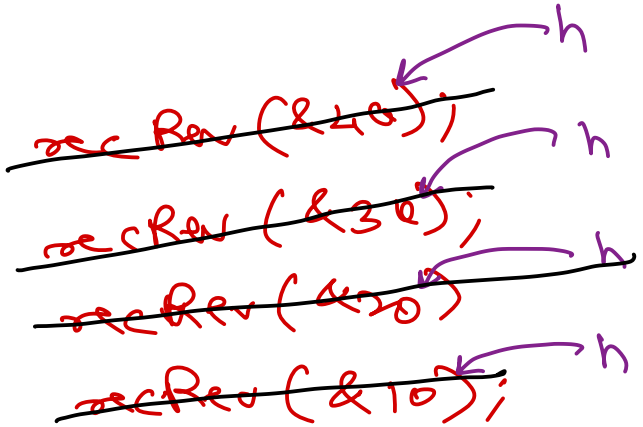
④ return cur node (last node);

Linked List – Competitive programming

- Reverse singly linked list using recursion.



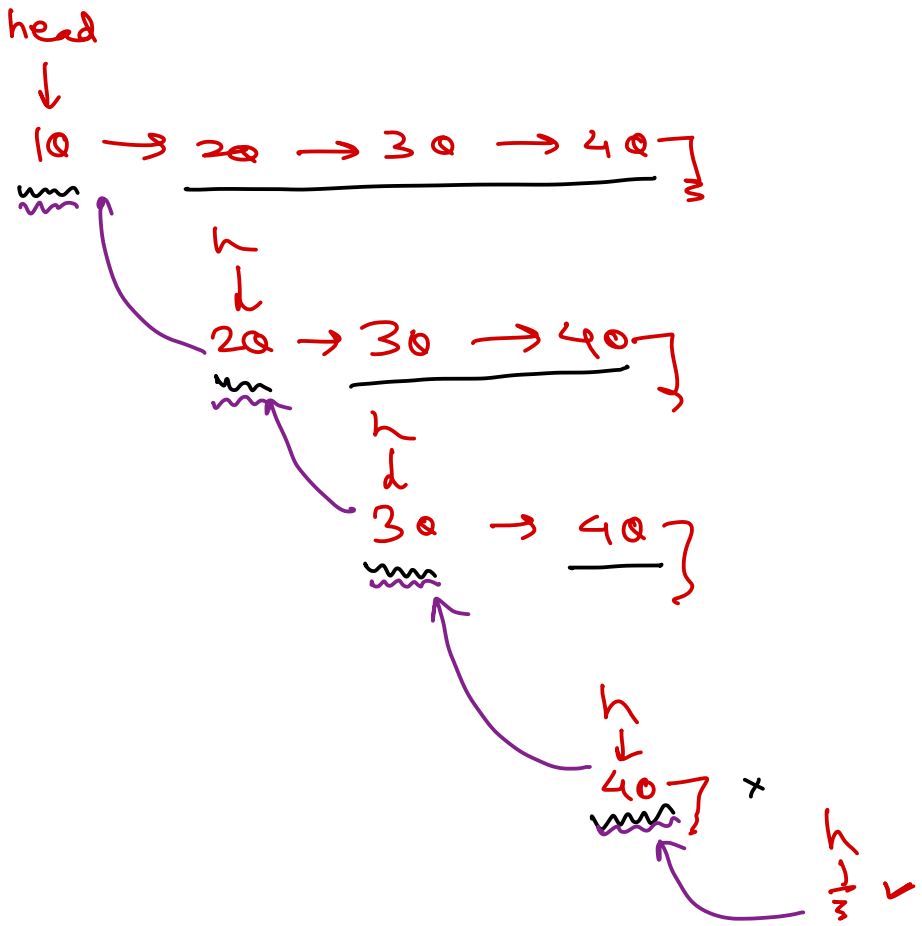

```
Node recRev(Node h){
    if(h.next==null){
        head = h;
        return h;
    }
    Node t= recRev(h.next);
    t.next = h;
    h.next=null;
    return h;
}
```



Linked List – Competitive programming

- ~~Reverse singly linked list using recursion.~~

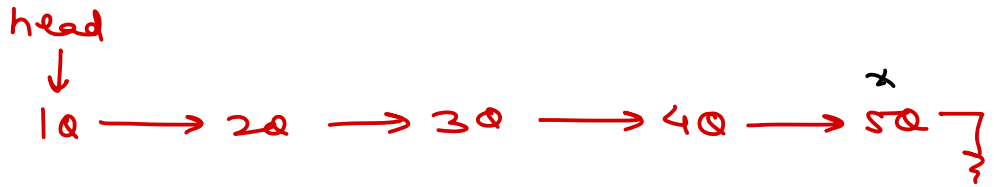
Display singly list in reverse order.

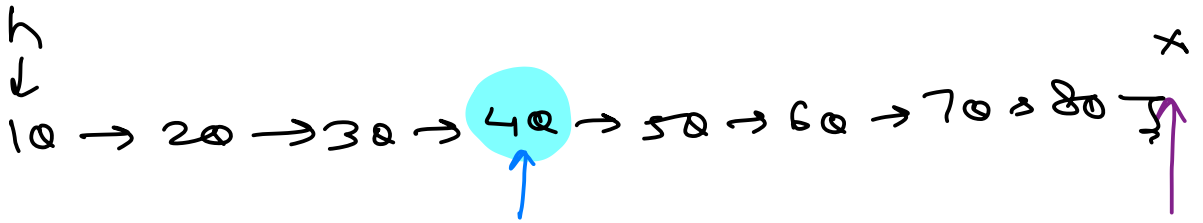



```
rev Display( h) {
→ if list is empty, return;
→ display rest of list (after recur);
→ display cur node;
}
```

Linked List – Competitive programming

- Find middle of singly linear linked list.





approach 1

1. traverse list & count num of nodes.
2. traverse till count/2.

approach 2

1. take two pointers - fast & slow.
2. traverse fast pointer: fast = fast.next.next;
3. traverse slow pointer: slow = slow.next;

```
fast = head;
slow = head;
while(fast != null && fast.next != null)
    slow = slow.next;
    fast = fast.next.next;
```

even nodes

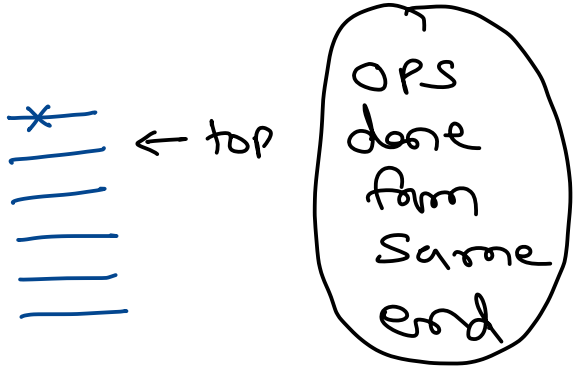
odd nodes

?

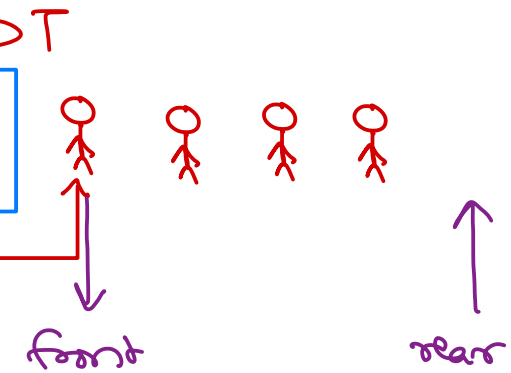
```
slow.data → return;
```

Stack and Queue

- Stack & Queue are utility data structures. → temp storage during process.
- Can be implemented using array or linked lists.
- Usually time complexity of stack & queue operations is O(1).
- Stack is Last-In-First-Out structure.
- Stack operations ↔ ADT
  - push()
  - pop()
  - peek()
  - isEmpty()
  - isFull()*



- Simple queue is First-In-First-Out structure.
- Queue operations – ADT
  - push() / enque()
  - pop() / deque()
  - peek()
  - isEmpty()
  - isFull()*

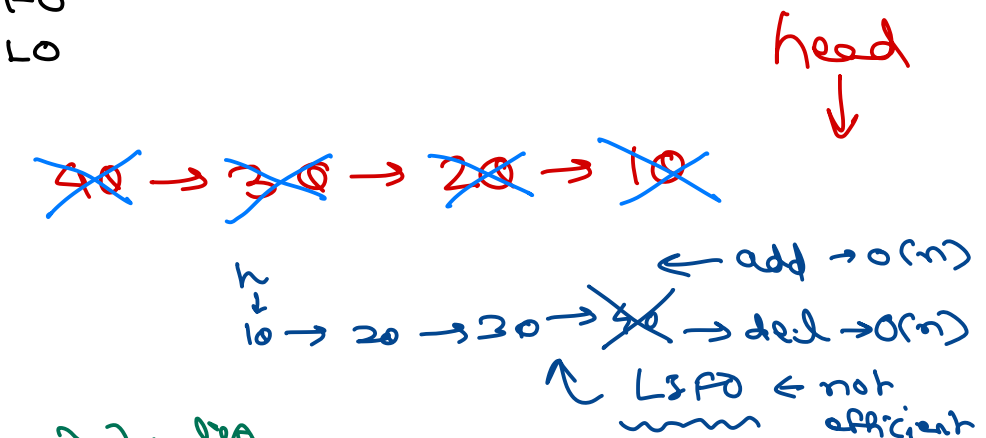


- Queue types
  - Linear queue
  - Circular queue
  - Deque
  - Priority queue

Stack / Queue using Linked List

- Stack can be implemented using linked list.
  - add first → push() → O(1)
  - delete first → pop() → O(1)
  - is empty → isEmpty() → O(1)
  - peek() → return head.data; → O(1)

LIFO
FILO



- Queue can be implemented using linked list.
  - add last → push()
  - delete first → pop()
  - is empty → isEmpty()
  - peek() → return head.data;

Singly list with head
inefficient → O(n)
O(1)
O(1)
O(1)

Singly list with head & tail ✓
O(1)
O(1)
O(1)
O(1)

FIFO
LILO

Stack - using array

push:

```
top++;
arr[top] = val;
```

peek:

```
return arr[top]
```

pop:

```
top--;
```

isFull:

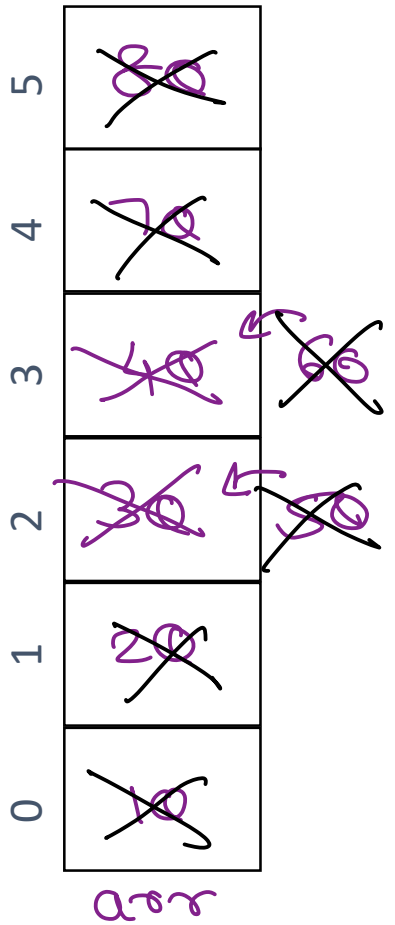
```
top == max-1
```

isEmpty:

```
top == -1
```

init:

```
top = -1;
arr = new int[];
```



top → -1

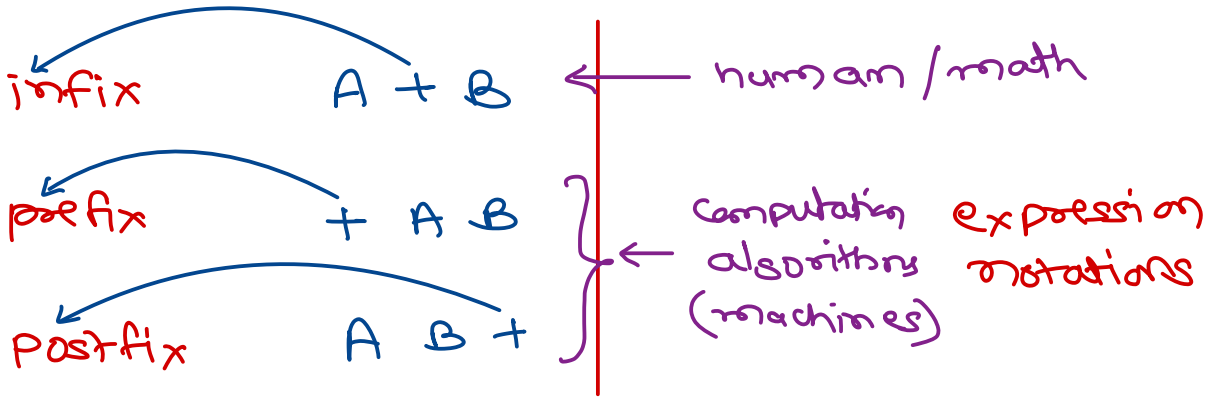
Stack / Queue in Java collections

- class java.util.Stack<E>
  - E push(E);
  - E pop();
  - E peek();
  - boolean isEmpty();

- interface java.util.Queue<E>
  - boolean offer(E e); - push
  - E poll(); - pop
  - E peek();
  - boolean isEmpty();

ArrayDeque<>
LinkedList<>

Expression notations.



infix → result ✓

OR

infix → postfix → result

infix → prefix → result

precedence

( )

$

* / %

+ −

← higher
←
←
← lowest

⑥ ⑦ ⑤ ② ① ⑧ ④ ③

5 + 9 − 4 * (8 − 6 / 2) + 1 $ (7 − 3)

5 + 9 − 4 * (8 − 6 2 /) + 1 $ (7 − 3)

5 + 9 − 4 * 8 6 2 / − + 1 $ (7 − 3)

5 + 9 − 4 * 8 6 2 / − + 1 $ 7 3 −

5 + 9 − 4 * 8 6 2 / − + 1 7 3 − $

5 + 9 − 4 8 6 2 / − * + 1 7 3 − $

5 9 + − 4 8 6 2 / − * + 1 7 3 − $

5 9 + 4 8 6 2 / − * − + 1 7 3 − $

5 9 + 4 8 6 2 / − * − 1 7 3 − $ +

⑥ ⑦ ⑤ ② ① ⑧ ④ ③

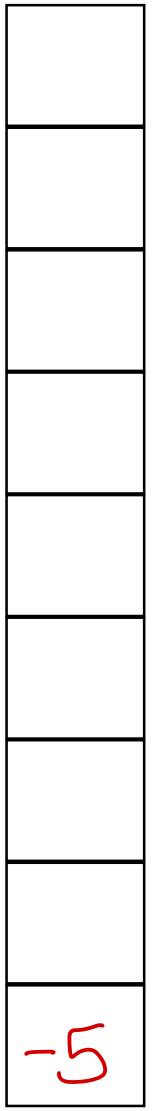
5 + 9 − 4 * (8 − 6 / 2) + 1 $ (7 − 3)

+ − + 5 9 * 4 − 8 / 6 2 $ 1 − 7 3

Postfix Evaluation

operands stack

- 5 9 + 4 8 6 2 / - * - 1 7 3 - $ + ↑



① traverse postfix from left to right.
② if sym is operand, push on stack.
③ if sym is operator, pop two args from stack, calc result & push on stack.
first poped - 2nd op / second poped - 1st op.
④ repeat 2 & 3 until all values from expr are done.
⑤ pop the final result from stack.

Thank you!

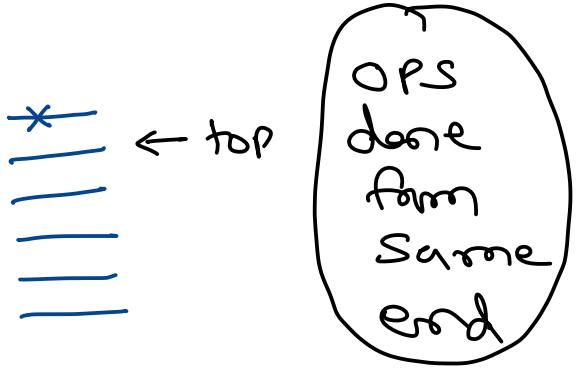
Nilesh Ghule <nilesh@sunbeaminfo.com>

Data Structure & Algorithms

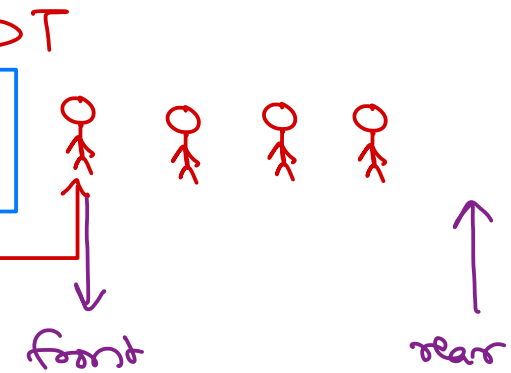
Sunbeam Infotech

Stack and Queue

- Stack & Queue are utility data structures. → temp storage during process.
- Can be implemented using array or linked lists.
- Usually time complexity of stack & queue operations is O(1).
- Stack is Last-In-First-Out structure.
- Stack operations ↔ ADT
  - push()
  - pop()
  - peek()
  - isEmpty()
  - isFull()*



- Simple queue is First-In-First-Out structure.
- Queue operations – ADT
  - push() / enque()
  - pop() / deque()
  - peek()
  - isEmpty()
  - isFull()*



- Queue types
  - Linear queue
  - Circular queue
  - Deque
  - Priority queue

Linear Queue

push → rear
pop ← front

```
init:
    arr = new int[5];
    f = -1;
    r = -1;

push:
    r++;
    arr[r] = val;

pop:
    f++;

peek:
    return arr[f+1];
```

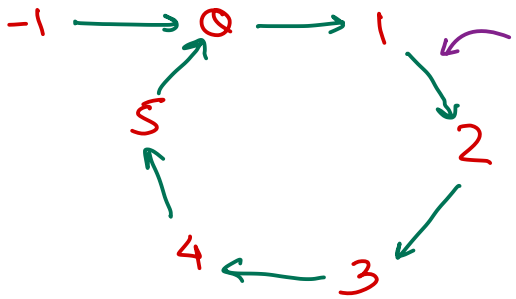
```
isFull:
    r == MAX-1

isEmpty:
    f == r
```

arr

0	1	2	3	4	5
10	20	30	40	50	60

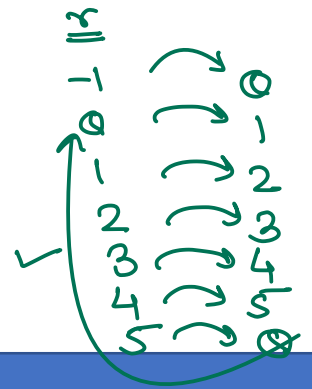
f = -1, r = 5

improper memory utilization



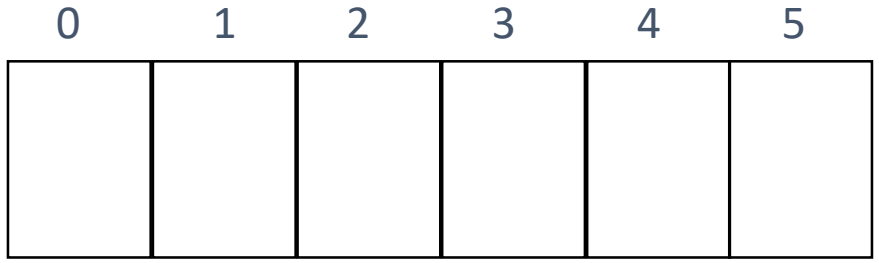

```
if (r == MAX-1)
    r = 0;
else
    r = r+1;
```

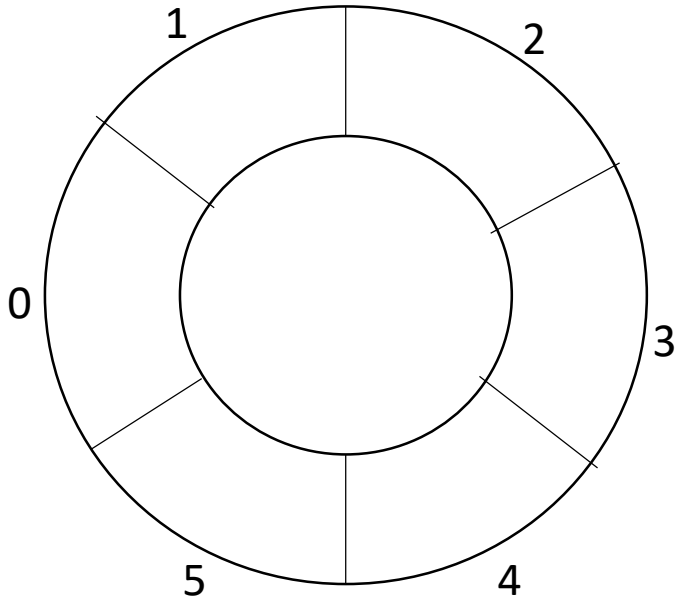
r = (r+1) % MAX;



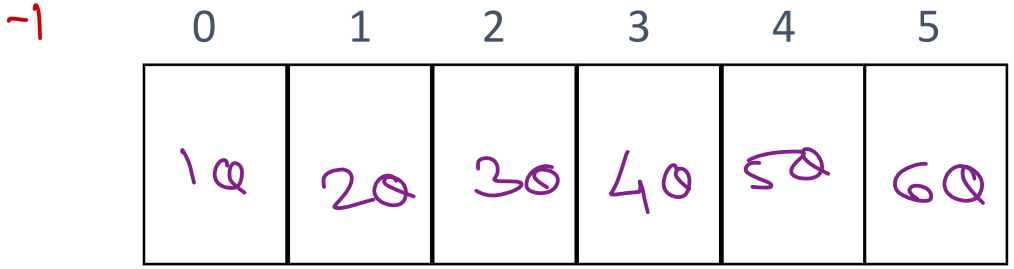
Circular Queue

- In linear queue (using array) when *rear* reaches last index, further elements cannot be added, even If space is available due to deletion of elements from *front*. Thus space utilization is poor.
- Circular queue allows adding elements at the start of array if *rear* reaches last index and space is free at the start of the array.
- Thus *rear* and *front* can be incremented in circular fashion i.e. 0, 1, 2, 3, …, n-1. So they are said to be circular queue.
- However queue full and empty conditions become tricky.

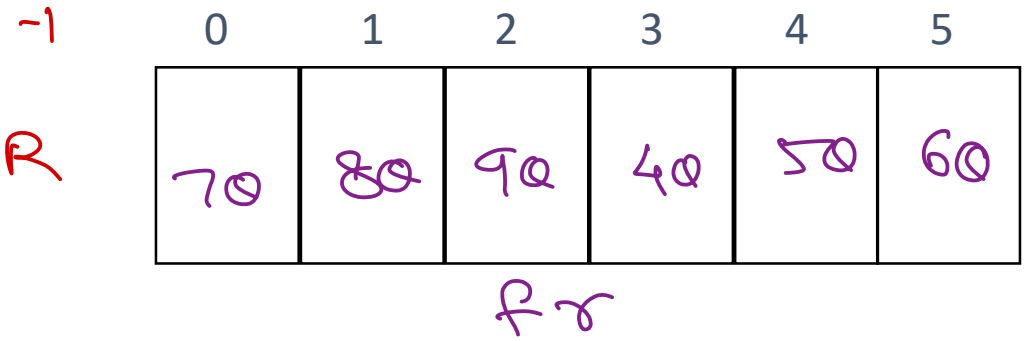




Circular Queue – full & push



OR



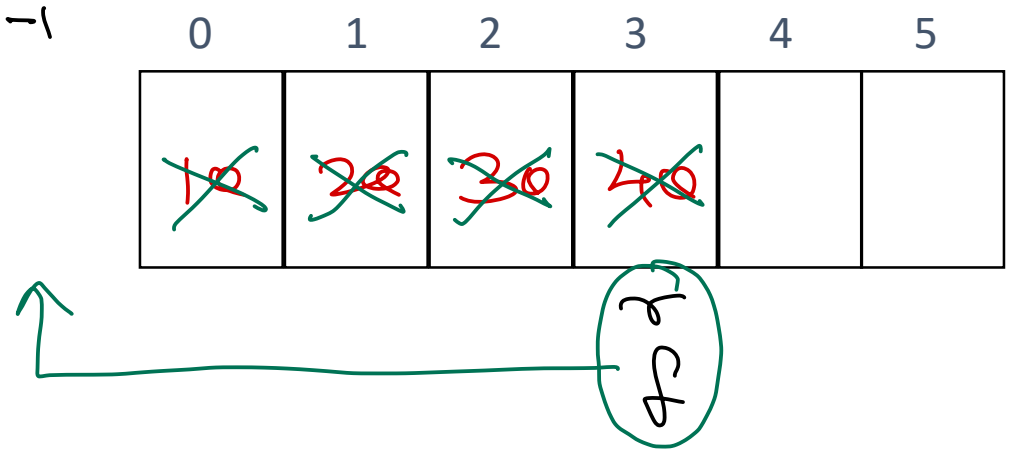

```
(f == -1 && r == MAX-1) || (f == r && f != -1)
```

push:

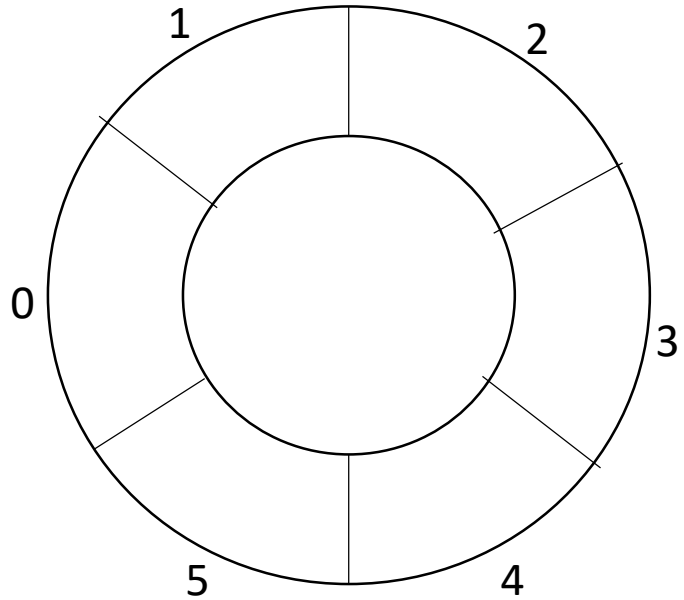
```
r = (r+1) % MAX;
arr[r] = val;
```

Circular Queue - empty & pop

```
( r == f && f == -1 )
```




```
pop :
    f = (f + 1) % max ;

    if( f == r )
    {   f = -1
        r = -1
    }
```

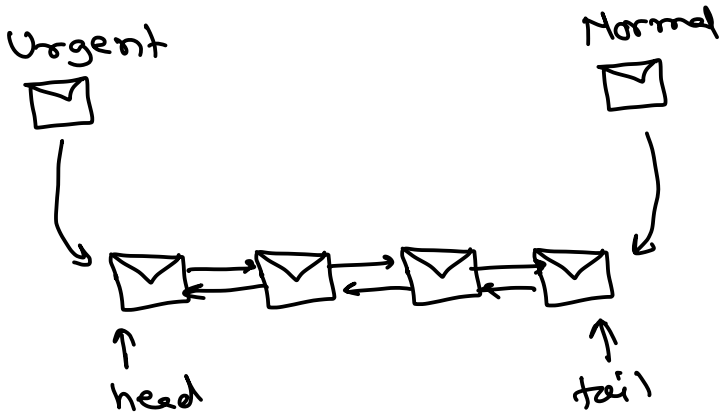


DeQueue

- In double ended queue, values can be added or deleted from front end or rear end.

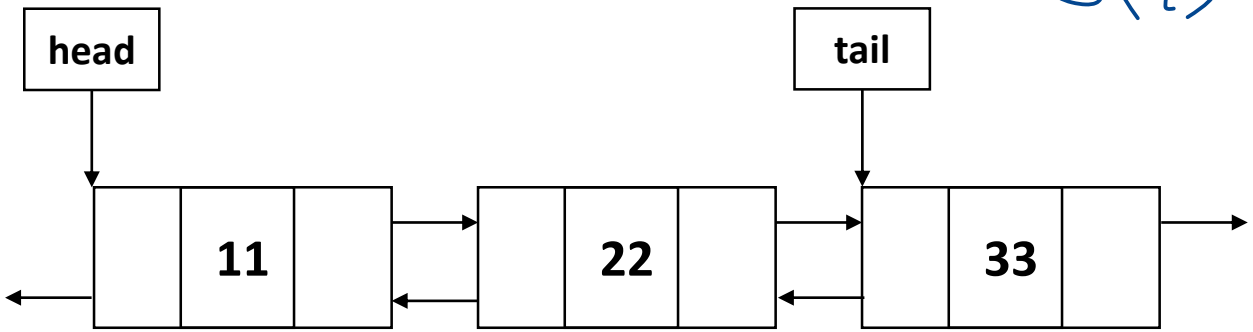
applications

message queue in RTOS.



O(1)
push_front
pop_front
O(1)

O(1)
push_back
pop_back
O(1)



Priority queue → Not a FIFO queue

- In priority queue, element with highest priority is removed first.

↳ internally elements are stored in sorted manner.

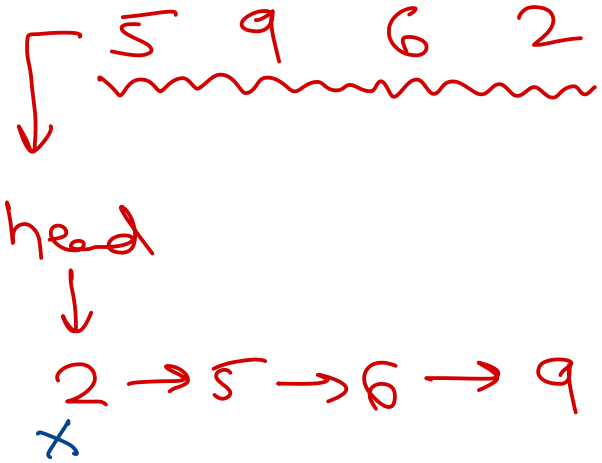
Can be implemented

① using array → insertion logic.

② using linked list → insertion logic.

push = O(n)

pop = O(1)



③ using heap (array representation of bin tree).

efficient

push = O(log n)

pop = O(log n)

Stack / Queue in Java collections

- class java.util.Stack<E>
  - E push(E);
  - E pop();
  - E peek();
  - boolean isEmpty();

- interface java.util.Queue<E>
  - boolean offer(E e); - push
  - E poll(); - pop
  - E peek();
  - boolean isEmpty();

ArrayDeque<>

LinkedList<>

PriorityQueue<>

Infix to Postfix

- 5 + 9 – 4 * (8 – 6 / 2) + 1 $ (7 – 3)

① traverse infix from left to right.

② if operand found, append to postfix.

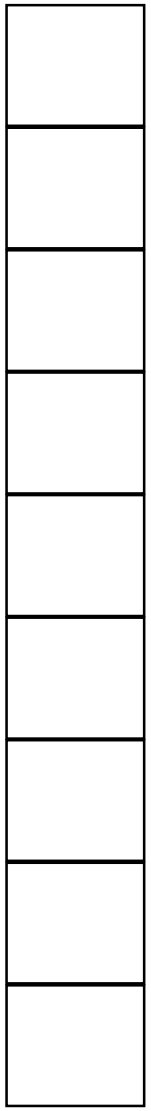
③ if operator found, push to stack ④.

  ④ if priority of topmost operator on stack >= priority of current operator, pop it from stack & append to postfix.

④ if opening (found, push it on stack.

⑤ if closing) found, pop operators from stack & append to postfix until opening is found. also pop & discard opening (from stack.

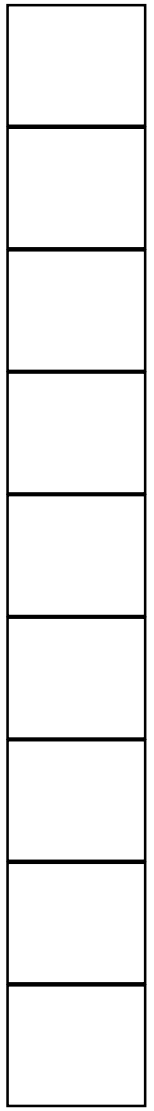
⑥ if all syms from infix are completed, pop one by one & append to postfix.



Infix to Prefix

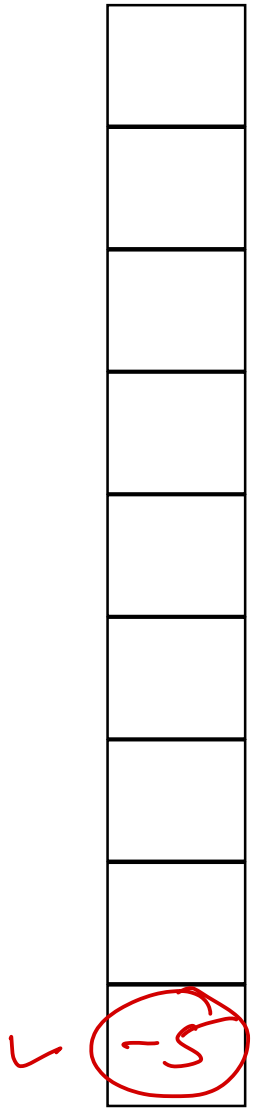
stack

- 5 + 9 – 4 * (8 – 6 / 2) + 1 $ (7 – 3)



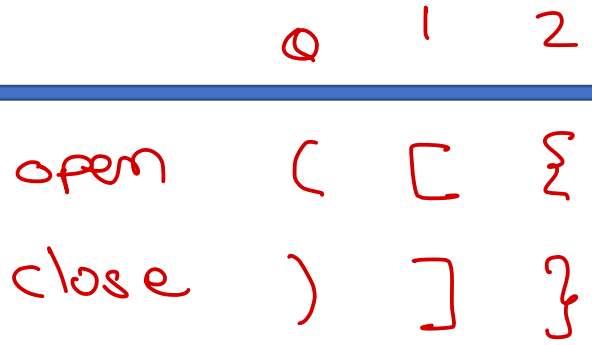
Prefix Evaluation

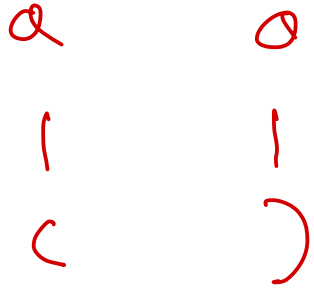
- + - + 5 9 * 4 – 8 / 6 2 $ 1 – 7 3

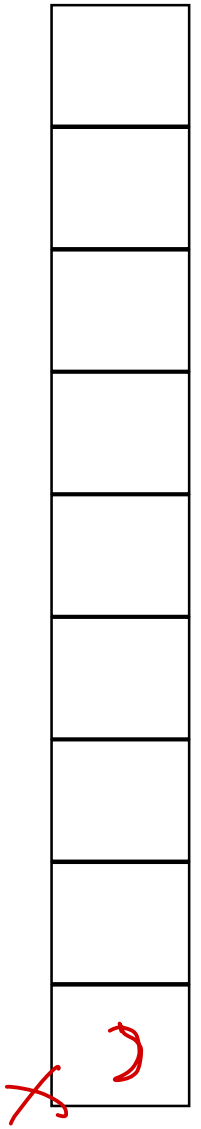


Parenthesis Balancing

- 5 + ([9 – 4] * (8 – { 6 / 2 }))

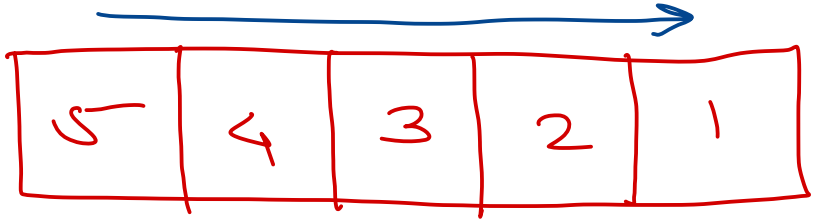






Stack / Queue – Competitive Programming

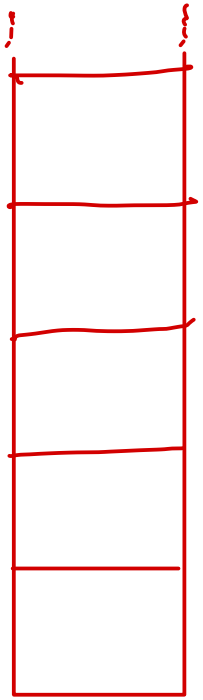
- Reverse array, string or linked list.



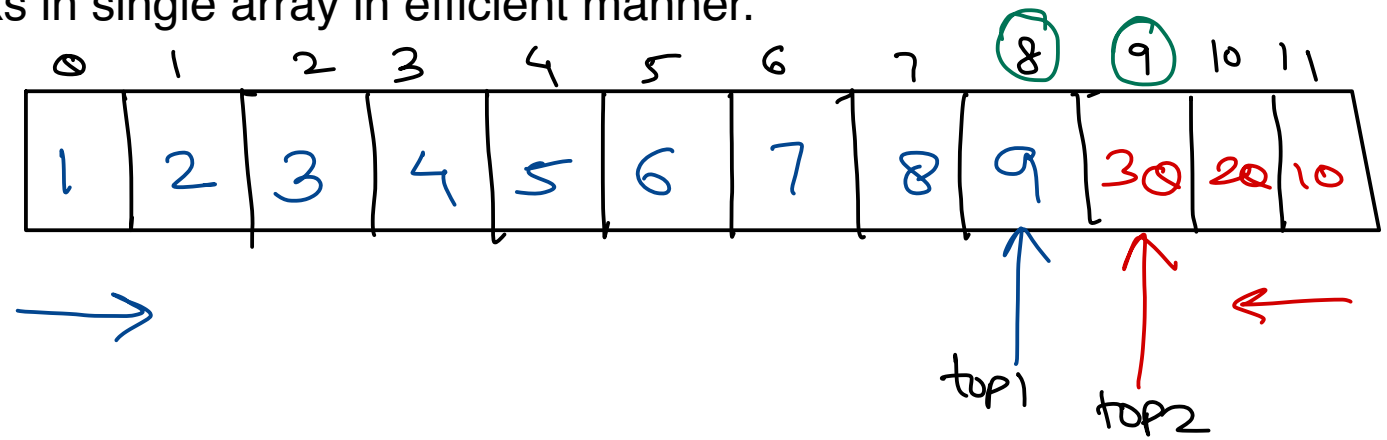

```
for(i=0; i<n; i++)
    s.push(arr[i]);
```
O(n)

```
for(i=0; i<n; i++)
    arr[i] = s.pop();
```
O(n)

O(2n) = O(n)



- Create two stacks in single array in efficient manner.




```
init1:
    top1 = -1
isEmpty1:
    top1 == -1
push1():
    top1++;
    arr[top1] = val;
pop1():
    top1--;
isFull():
    top1 + 1 == top2
```

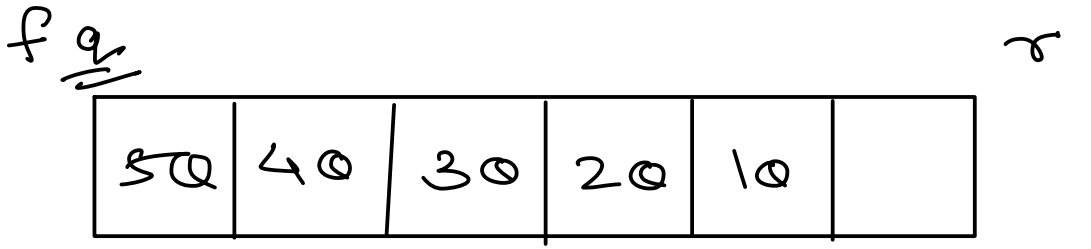
- top1 + 1 == top2
- top1 == top2 - 1
- top2 - top1 == 1

```
init2:
    top2 = arr.length
isEmpty2:
    top2 == arr.length;
push2():
    top2--;
    arr[top2] = val;
pop2():
    top2++
isFull:
    top1 + 1 == top2
```

Stack / Queue – Competitive Programming

- Create stack using queue.

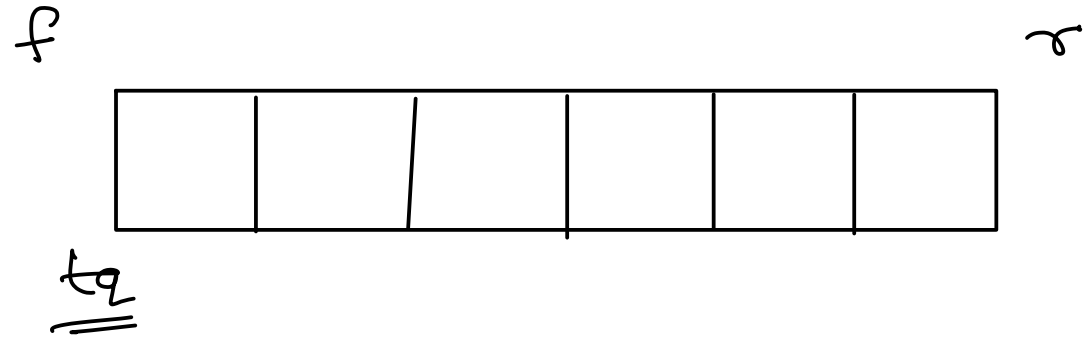
LIFO FIFO



50 →

push → O(n)

```
tq
while(!q.isEmpty()){
    tq.push(q.pop());
}
q.push(val);
while(!tq.isEmpty()){
    q.push(tq.pop());
}
```



pop → O(1)

q.pop()

Stack / Queue – Competitive Programming

- Create queue using stack.

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>

Data Structure & Algorithms

Sunbeam Infotech

Hash Table – Searching

Hash Table is DS in which data is stored in key-value pair, so that for a given key value can be searched in fastest possible time. Ideal is O(1).

Key-value pair → associative DS

e.g. Mobile → Contacts appln → name (key) ⇒ mobile (value)

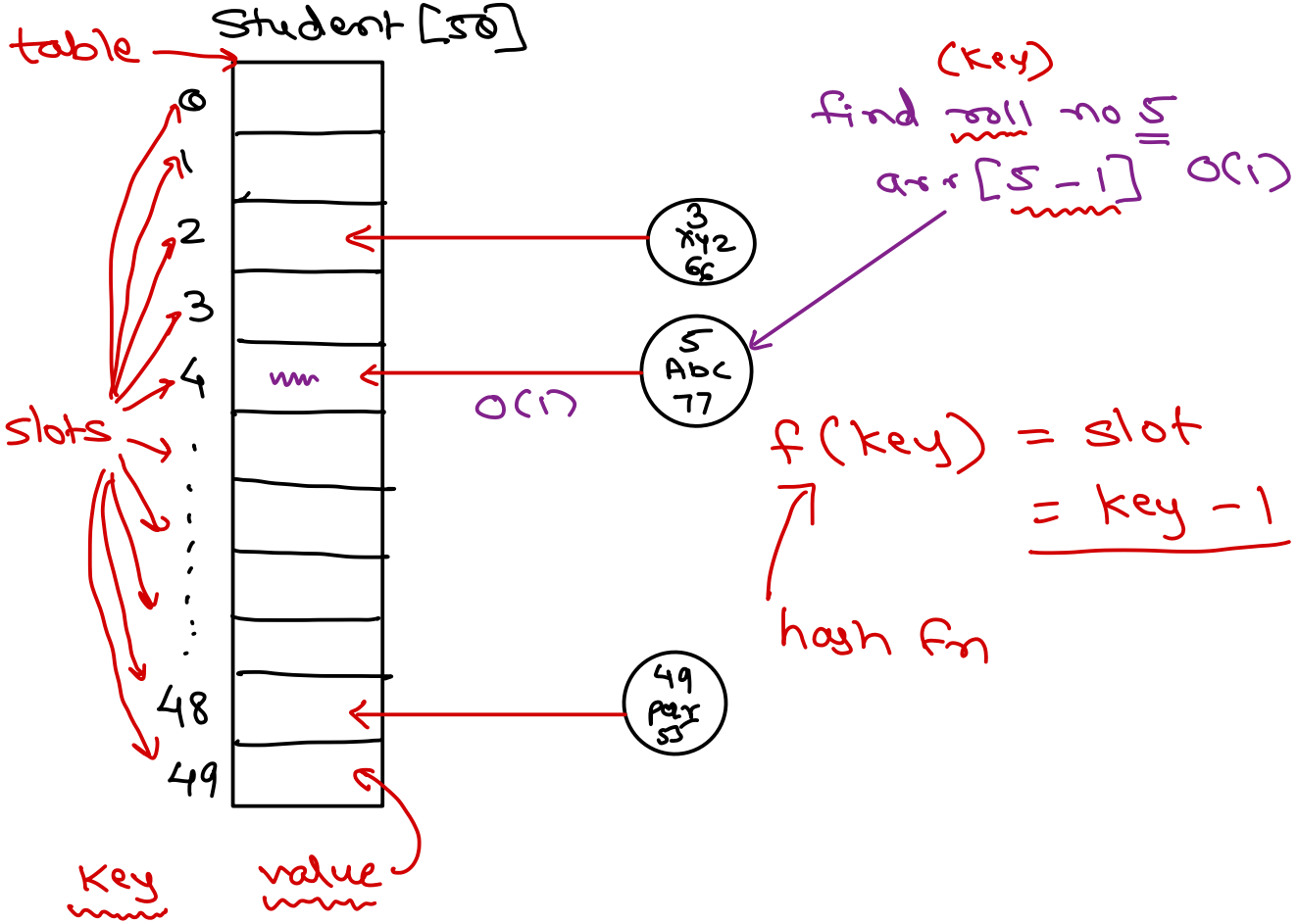
Hash ADT

1. put(key, value)
2. value = get(key)

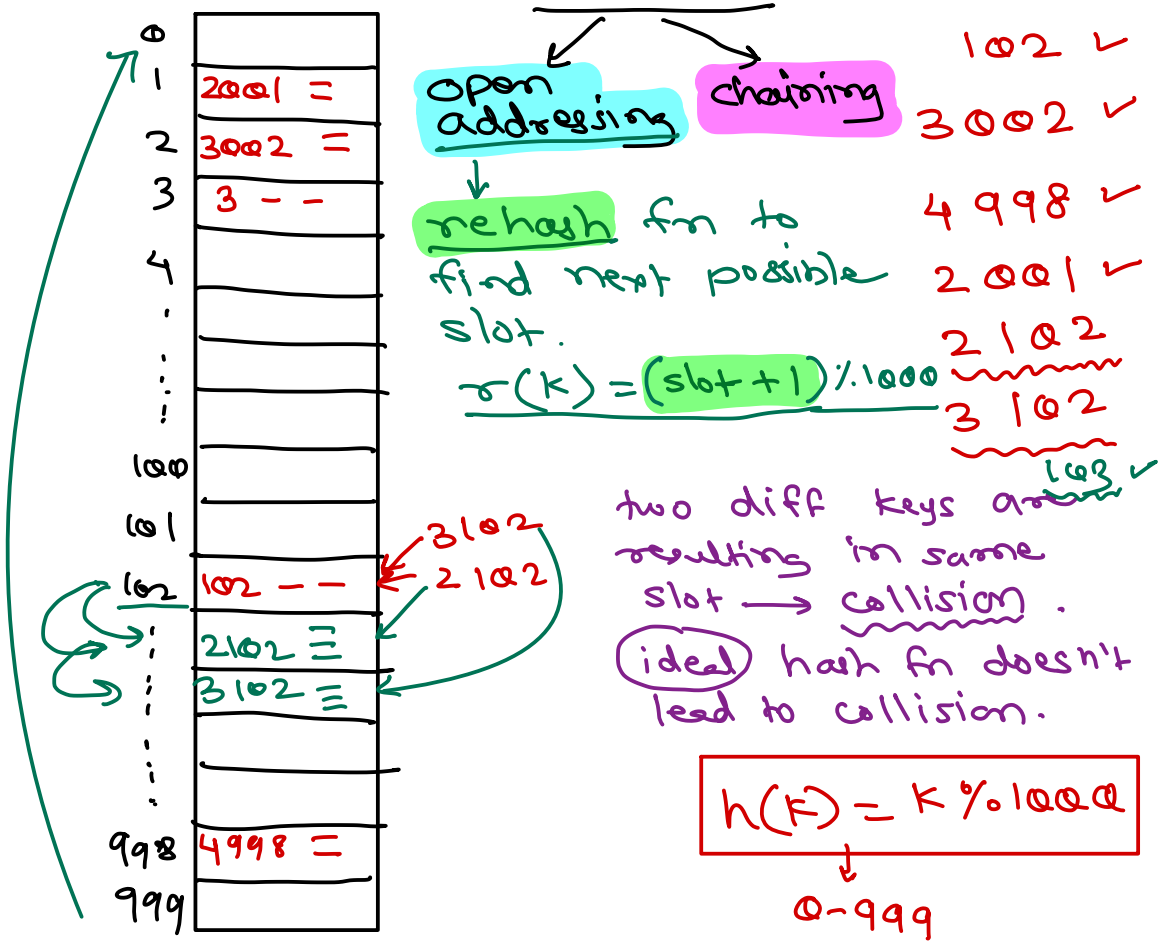
Searching

1. Linear → O(n)
2. Binary → O(log n)
3. ~~Fibbonacci~~
4. Hashing → O(1) ideal

A class have 50 students with roll 1 to 50.
Store students so that, student can be found in fastest way for given roll.



in an alumni meet max 1000 students will come. invitations sent to 5000 students. Store details so that given prn num, student can be found quickly.



Open Addressing

① if collision occur, call rehash fn.

② rehash fn find next possible slot to store/find the element → probing.

③ rehash fn
- linear – $ax+b$
- quadratic – ax^2+bx+c
- polynomial

④ find process:
- ⓐ hash fn → slot
- ⓑ check if ele found in the slot.
- ⓒ if not call rehash fn.
- ⓓ repeat b & c until ele is found.

⑤ open addr mechanism is internal storage. Data stored within array/table.

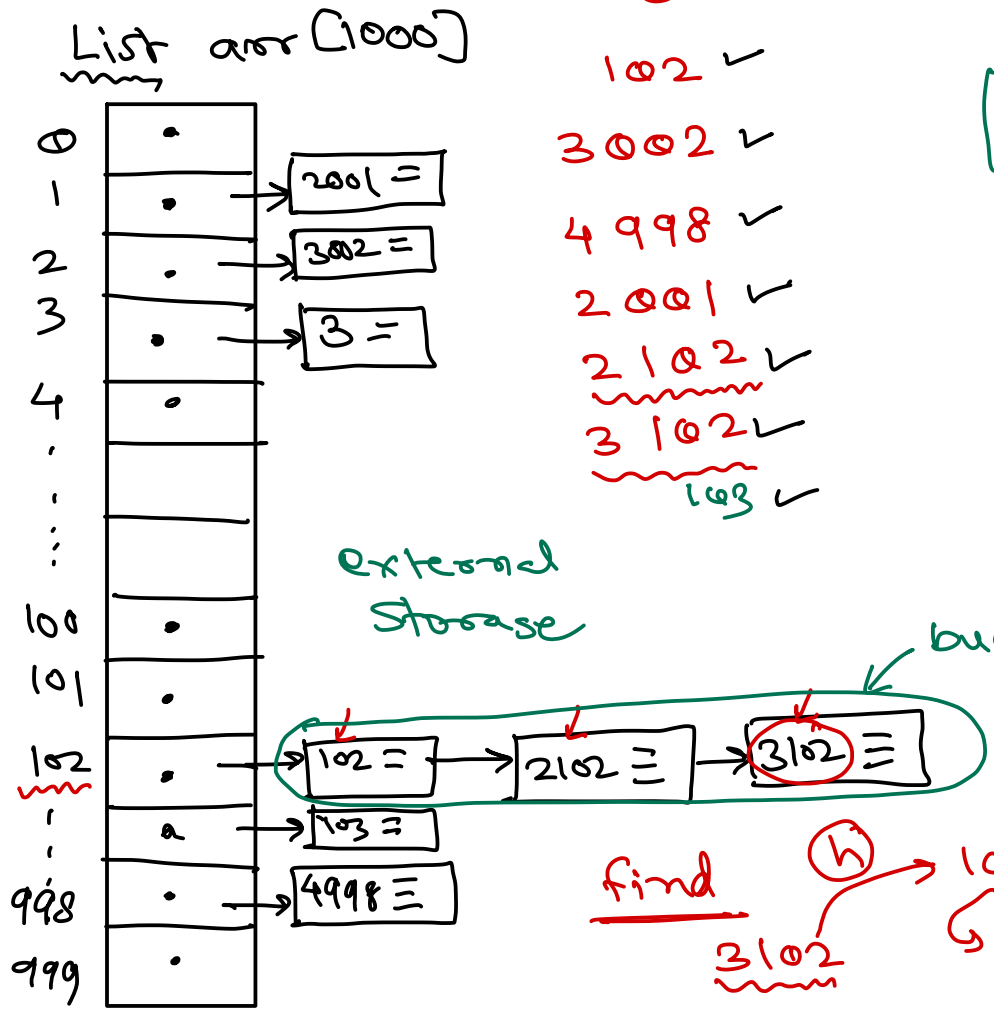
Load factor

$$= \frac{\text{num of kv pairs}}{\text{num of slots}}$$

* num of kv pairs < num of slots
  * load factor < 1
* num of kv pairs = num of slots
  * load factor = 1
* num of kv pairs > num of slots
  * load factor > 1

→ open addressing cannot be used.

chaining



Java collections : HashMap, ...
- chaining method

$$h(k) = 100$$

1. more the num of collisions, slower is the performance for add & search.
2. to reduce collisions better (uniform) hash fn should be implemented.
3. it is observed that multiply by prime num makes more uniform hash fn.

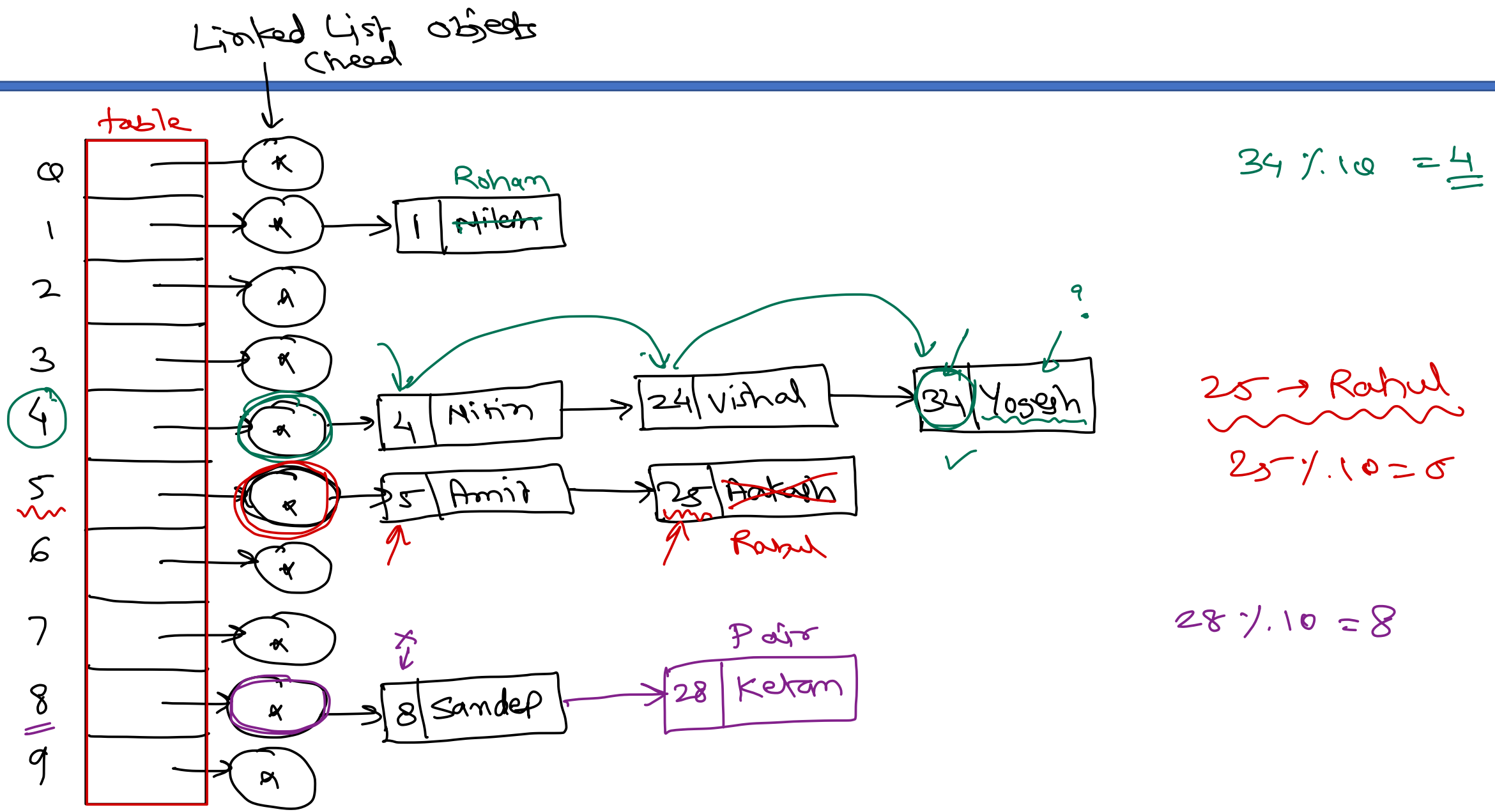
e.g. h(key) = (key * 31) % slots.

e.g. store students as per their height

key = height, value = student

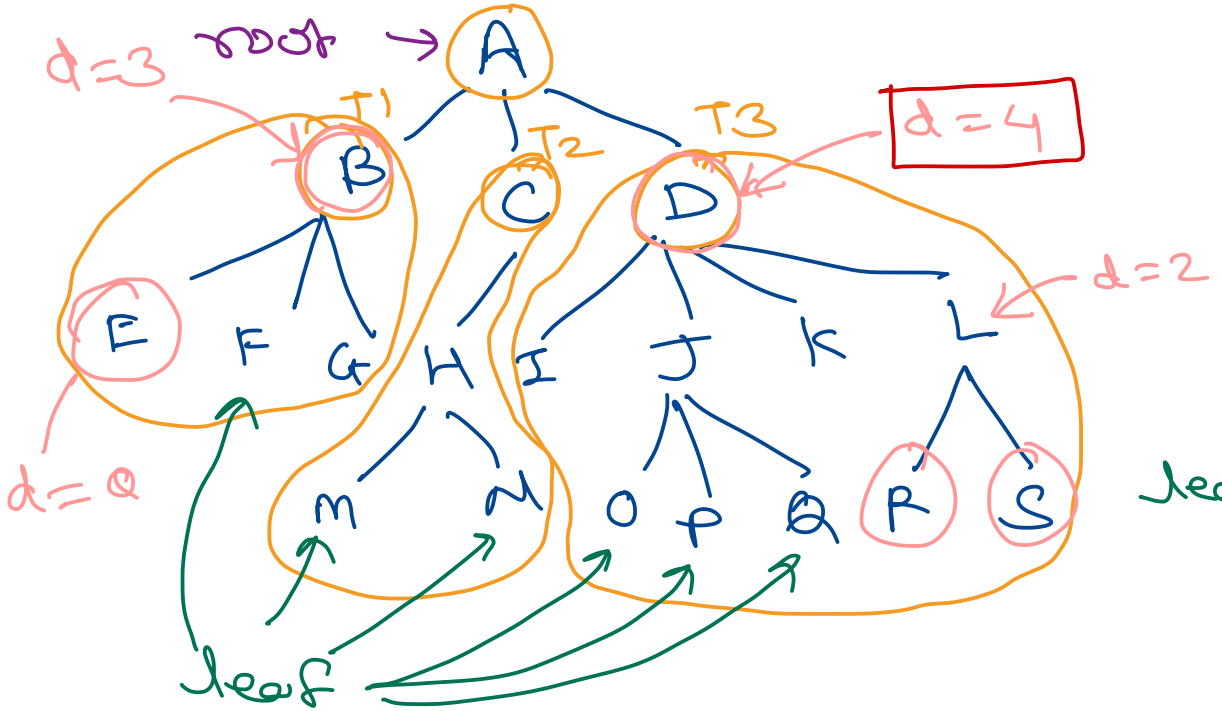
feet/inches.

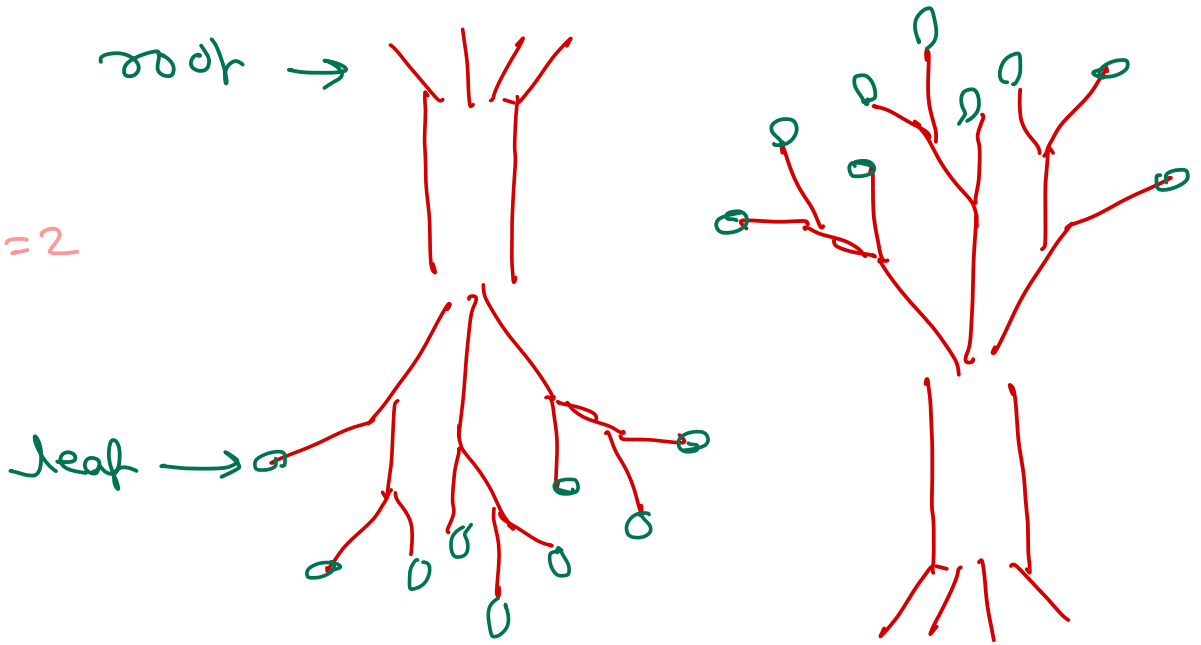
h(key) = (feet * 31 + inches * 31) % slots;


Linked List objects
Chead
table
0
1
2
3
4
5
6
7
8
9
Roham
1 Nilam
4 Nitin
24 Vishal
34 Yogesh
5 Amit
25 Rahul
Rahul
8 Sandep
28 Ketan
Pair
34 % 10 = 4
25 → Rahul
25 % 10 = 5
28 % 10 = 8

Tree Definition

- **Tree** is a finite set of nodes with one specially designated node called the "**root**" and the remaining node are partitioned into disjoints sets T1 to Tn, where each of those sets is a TREE.
- T1 to Tn are called **sub-trees** of the root



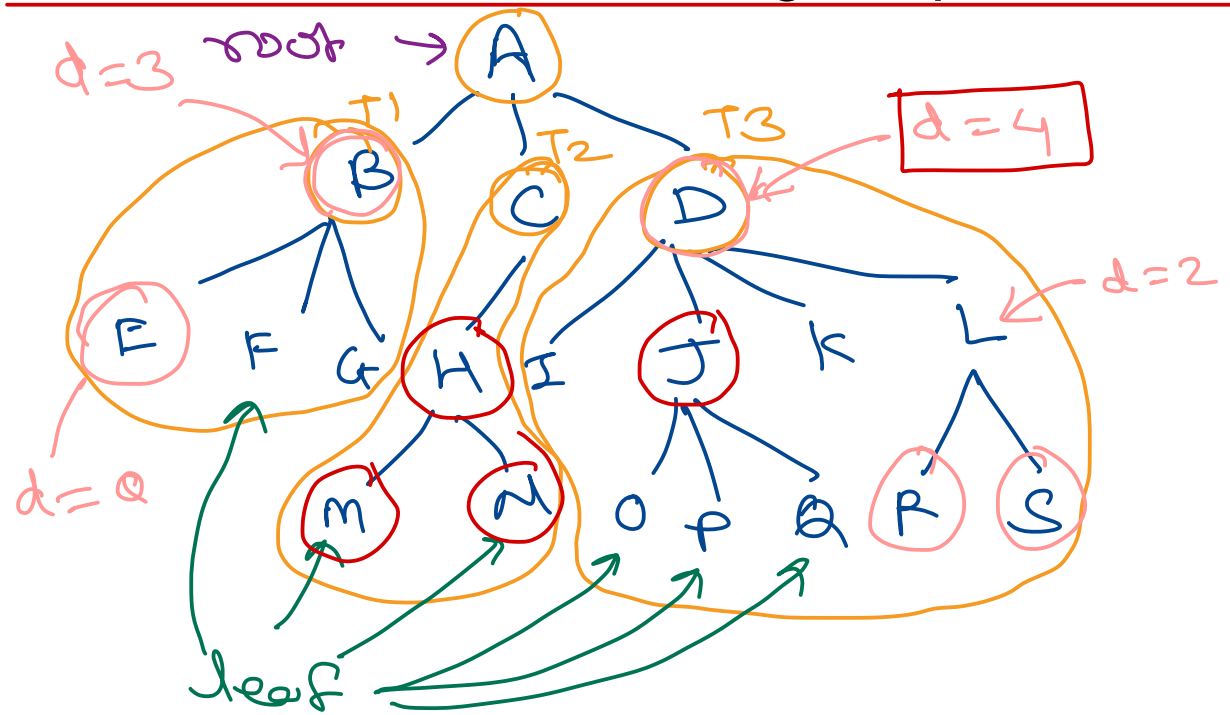


Tree terminologies

- Node: A item storing information and branches to other nodes
- Null Tree: Tree with no node
- Leaf Node: Terminal node of a tree & does not have any node connected to it
- Degree of a Node: No of sub trees of a node
- Degree of a tree: Degree of a tree is maximum degree of a node in the tree

Tree terminologies

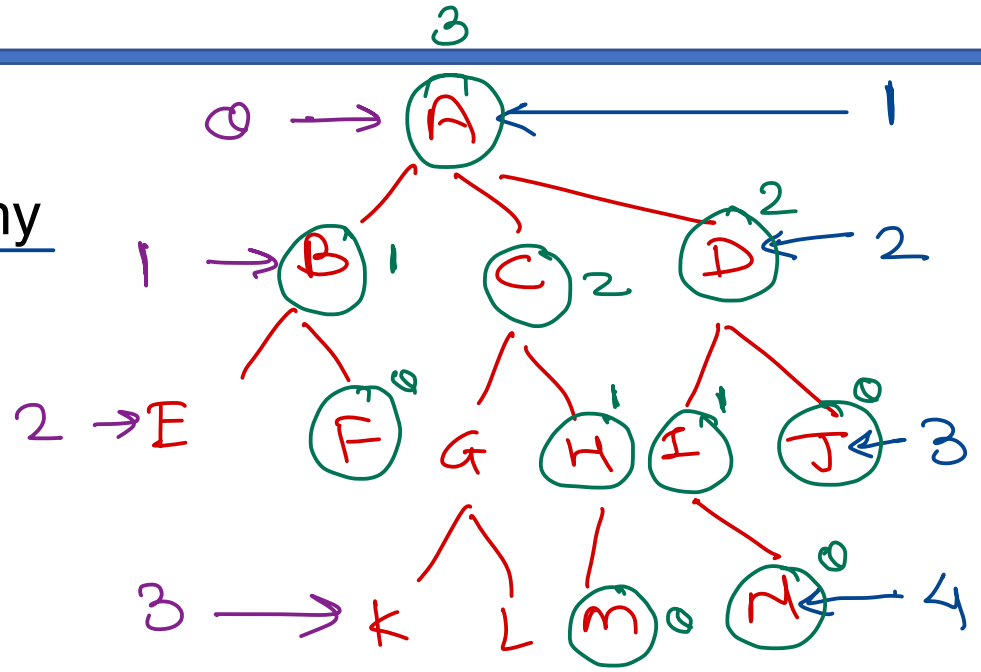
- Parent Node: node having other nodes connected to it
- Siblings: Children of the same parents → OPQ , BCD , MN , ...
- Descendants: all those node which are reachable from that node J → O P Q
- Ancestor: all the node along the path from the root to that node C → H M N



M → H , C , A
Q → J , D , A

Tree terminologies → can traversal

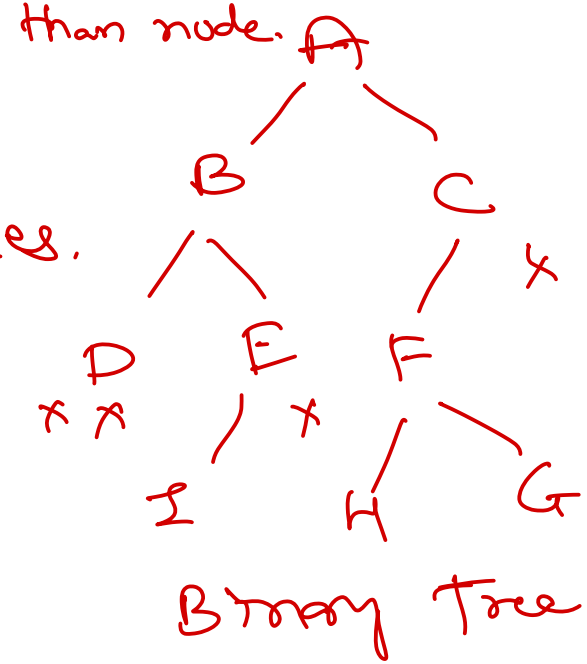
- Level of a Node:
  - Indicates the position of the node in the hierarchy
  - Level of any node is level of its parent +1
  - Level of root is 1
- Depth of a node:
  - Number of nodes from the root to the node.
  - Depth of root is 0
  - Level = Depth + 1
- Height of a node:
  - Number of nodes from the node to its deepest leaf.
  - Height of node = height of its child + 1
  - Height of empty/null tree is -1
- Height of a tree: Height of root of the tree.
- Traversal: Visiting each node of tree exactly once



Types of trees

- Binary Trees → max 2 child nodes for each node.
  - It is a finite set of nodes partitioned into three sub sets:- Root, Left sub tree, Right sub tree
- Binary Search tree
  - A binary search tree is a binary tree in which the nodes are arranged according to their values.

left child smaller than node
right child greater than node or equal



Multiway tree
or n-way tree.

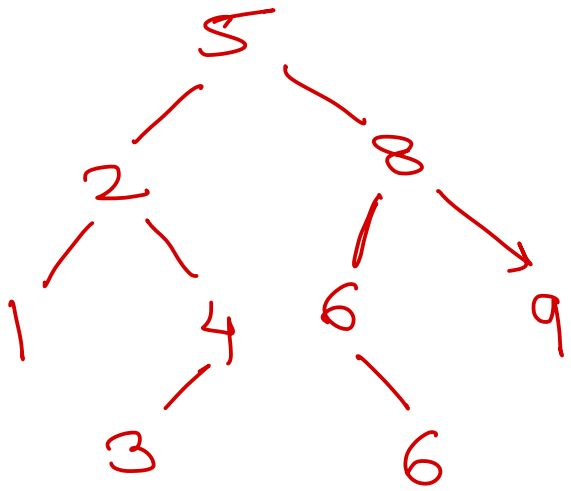
↓
each node have multiple child nodes.

```
class Node {
    int data;
    List<Node> children;
    =
    =
}
```

```
class Node {
    int data;
    Node left;
    Node right;
    =
}
```

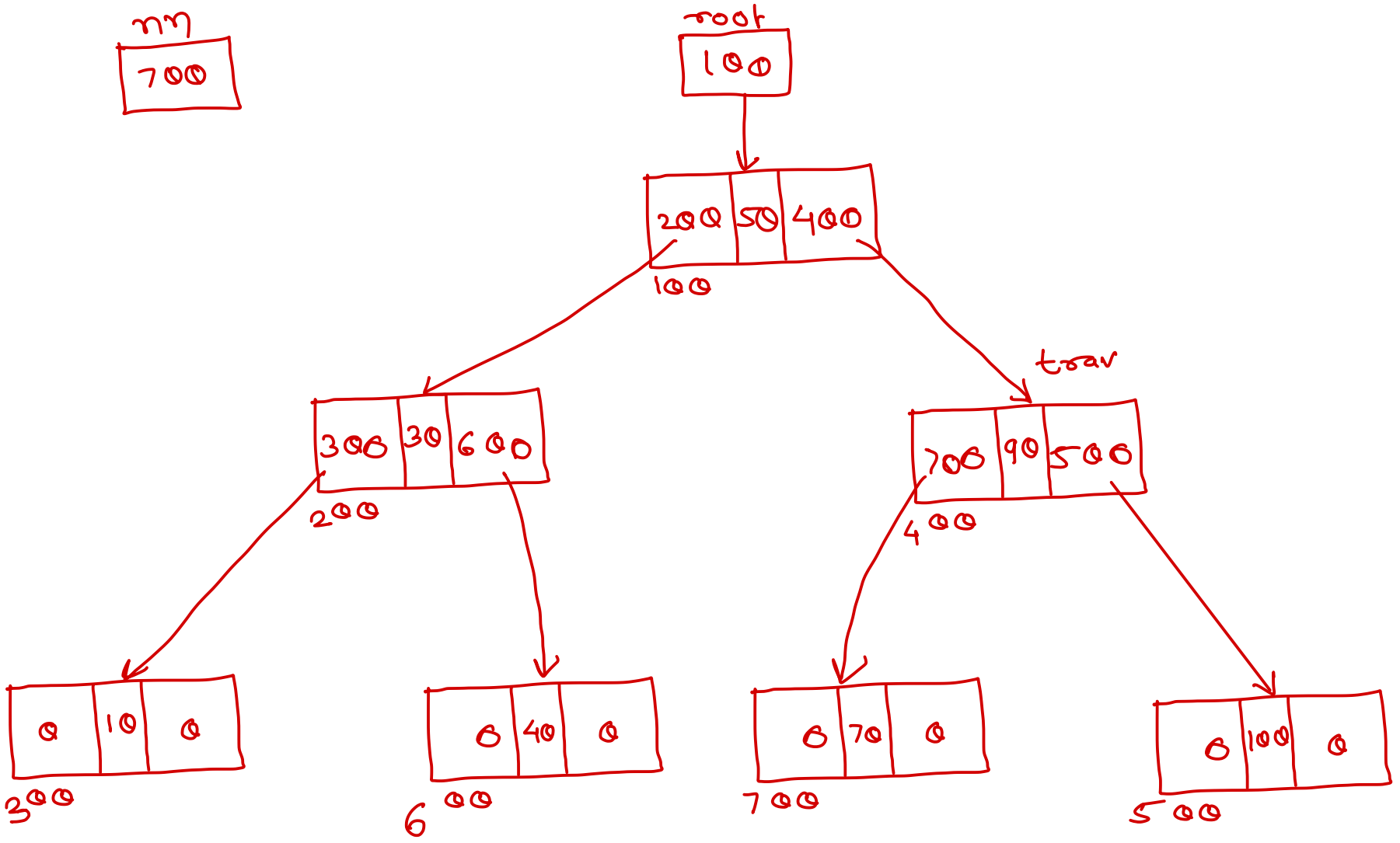
BST

5, 8, 2, 4, 6, 9, 1, 3, 6



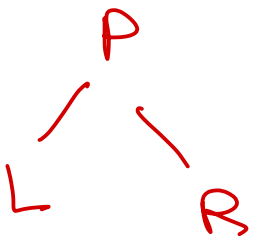
BST

50, 30, 10, 90, 100, 40, 70, 80, 60, 20



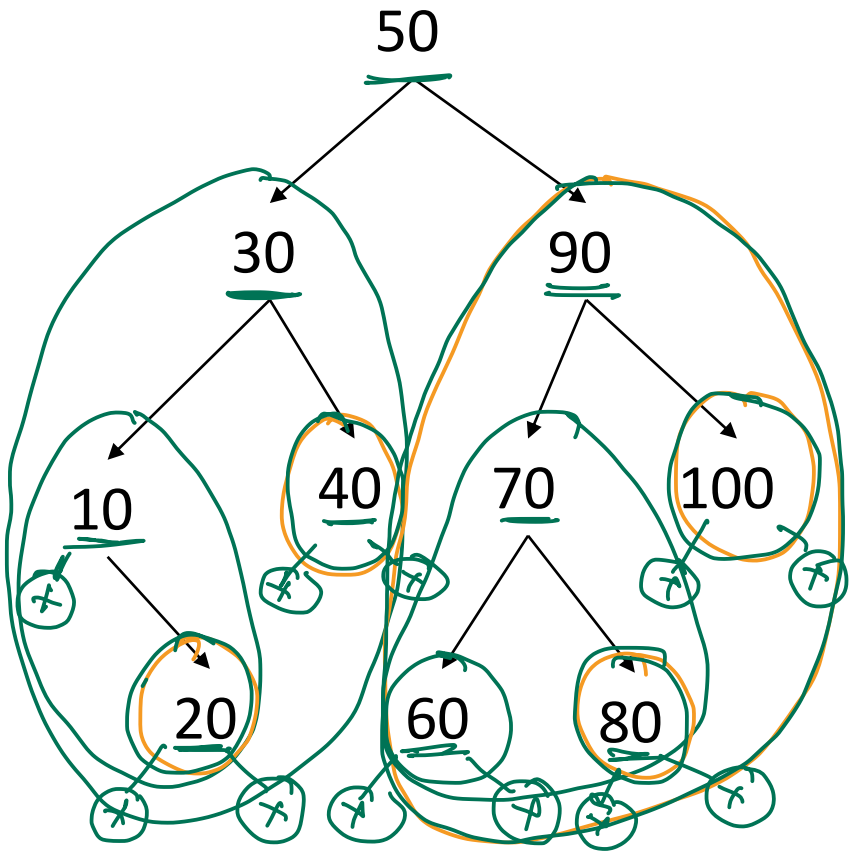
Binary Tree Traversal

- In-order: L P R
- Pre-Order: P L R
- Post-Order: L R P
- The traversal algorithms can be implemented easily using recursion.
- Non-recursive algorithms for implementing traversal needs stack to store node pointers.




```
preorder (trav).
    if(trav==null)
        return;
    print (trav.data);
    preorder(trav.left);
    preorder(trav.right);
```

BST → preorder – recursive.

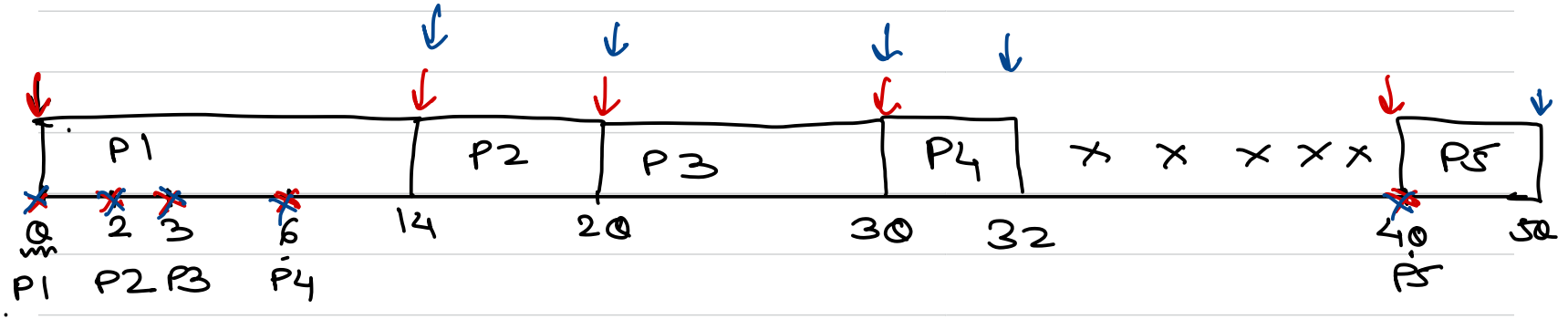



```
preorder (trav).
    if(trav == null)
        return;
    print (trav.data);
    preorder(trav.left);
    preorder(trav.right);
```

50 30 10 20 40

90 70 60 80 100

Assign 9



	waiting	turn-around
P1	0 − 0 = 0	14 − 0 = 14
P2	14 − 2 = 12	20 − 2 = 18
P3	20 − 3 = 17	30 − 3 = 27
P4	30 − 6 = 24	32 − 6 = 26
P5	40 − 40 = 0	50 − 40 = 10

class Job {
- ✓ id;
- ✓ arrival;
- ✓ burst;
- ? waiting;
- ? turnaroun

}

Queue<Job> ...

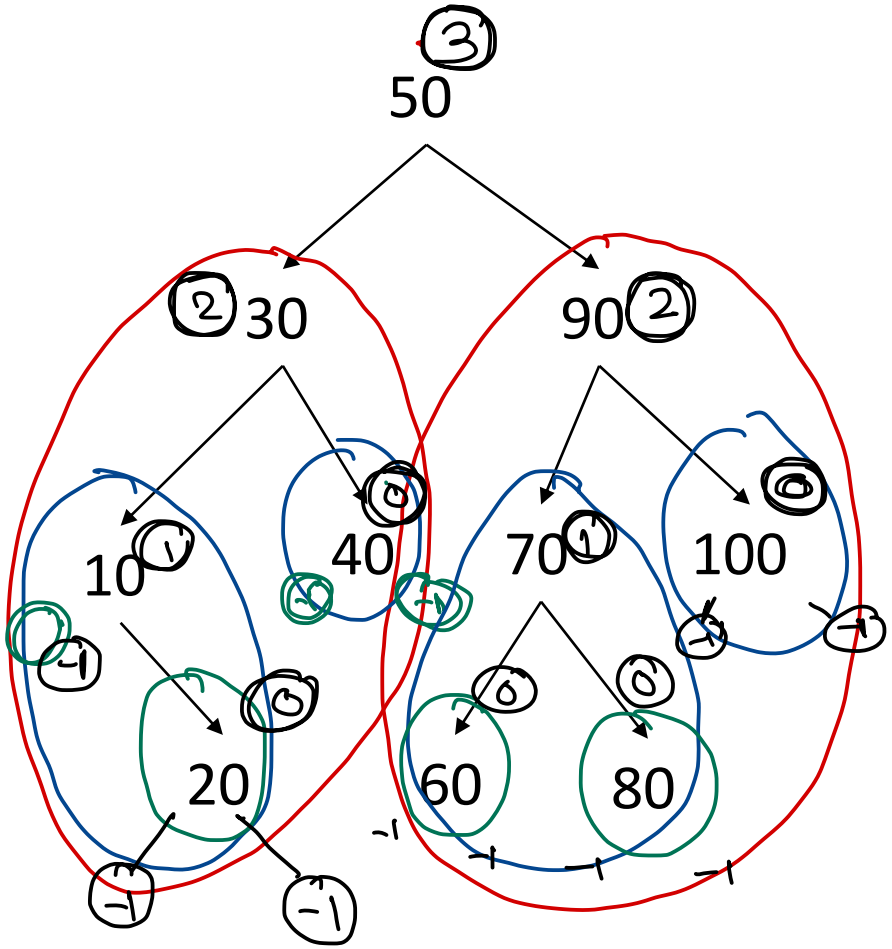
Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>

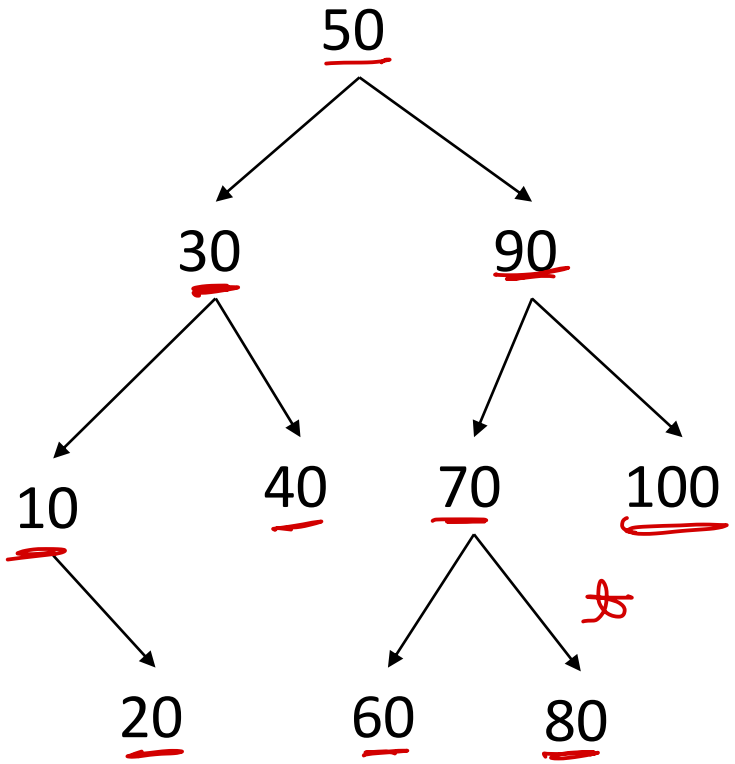
Data Structure & Algorithms

Sunbeam Infotech

BST - height()

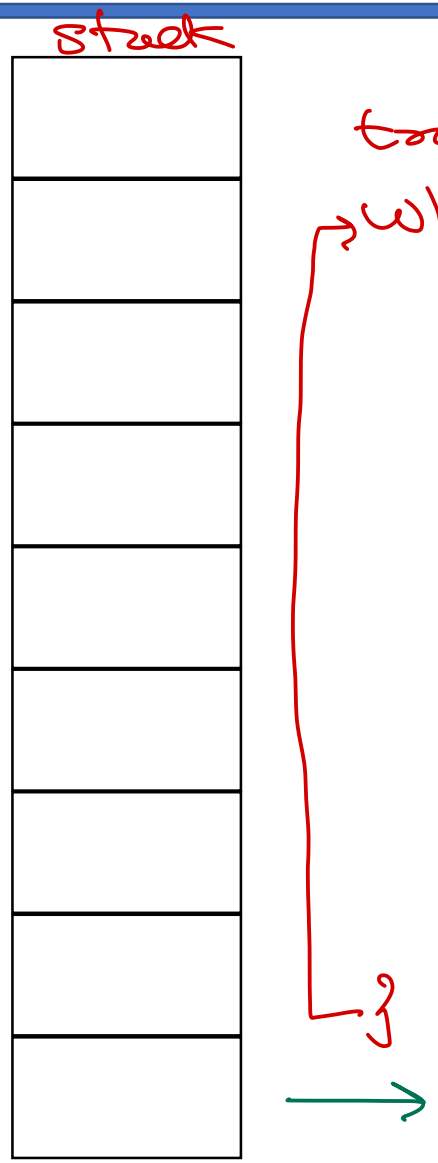


BST - preorder



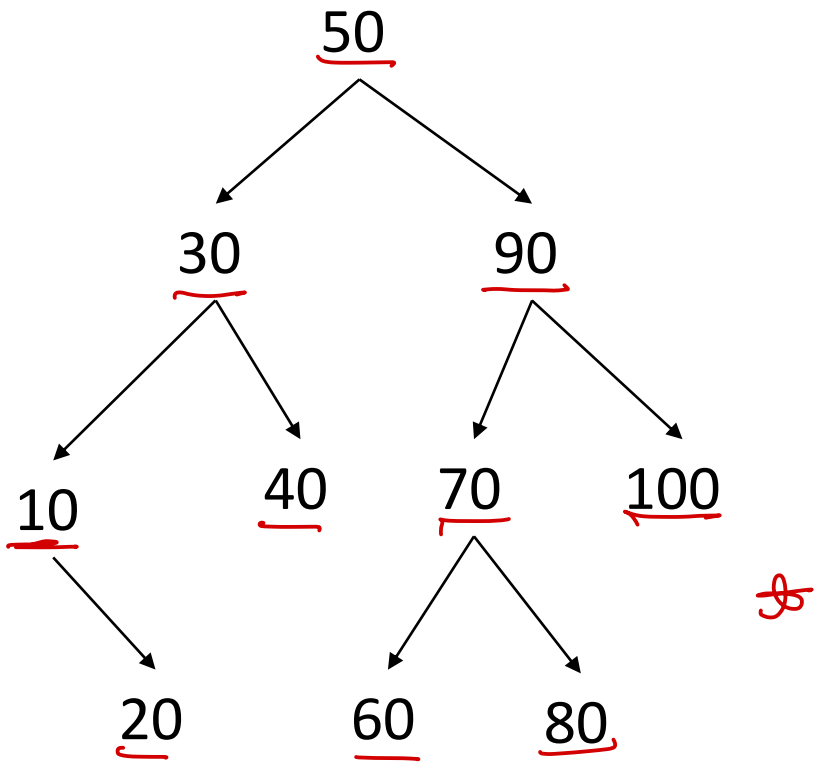
50 30 10 20 40

90 70 60 80 100

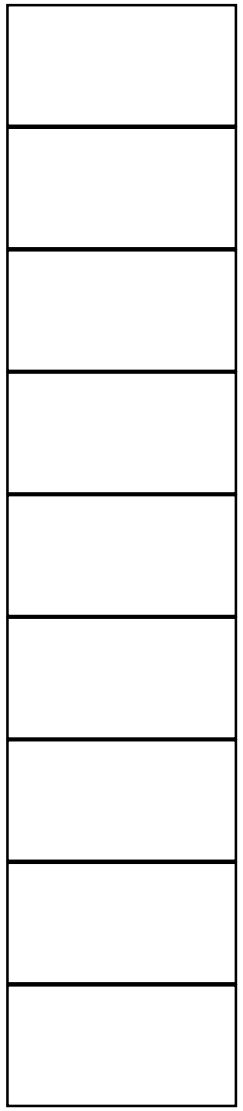



```
trav = root;
while(trav != null || !s.isEmpty()) {
    while(trav != null) {
        print(trav.data);
        if(trav.right != null)
            s.push(trav.right);
        trav = trav.left;
    }
    if(!s.isEmpty())
        trav = s.pop();
}
```

BST - inorder

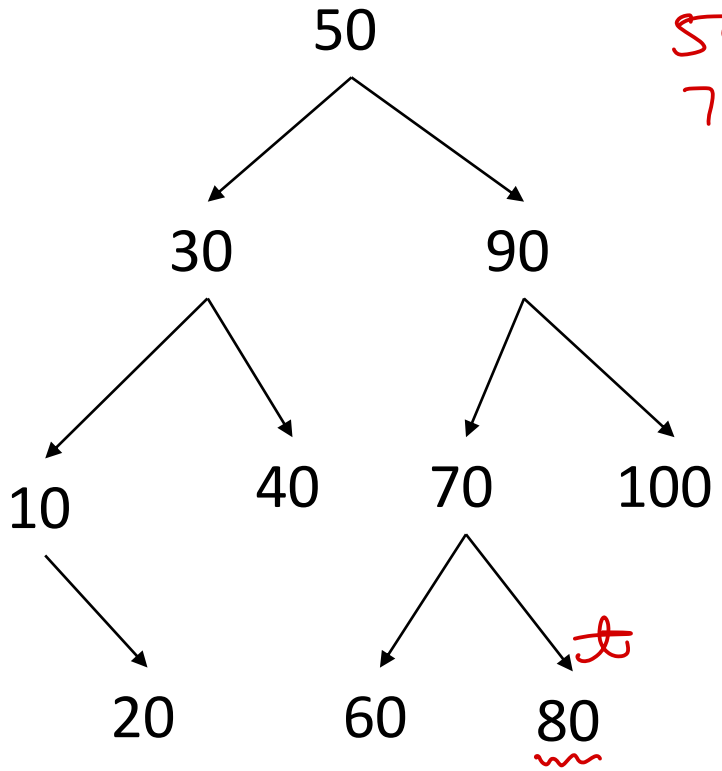


10 20 30 40 50
60 70 80 90 100



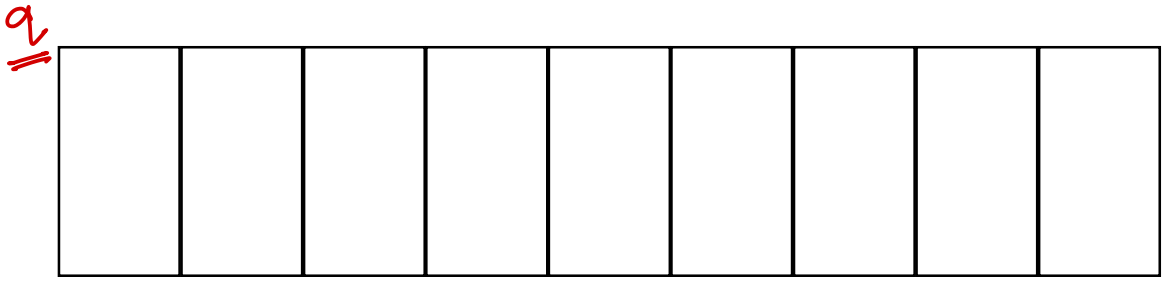
```
trav = root;
while(trav!=null || !s.isEmpty()) {

    while(trav!=null) {
        s.push(trav)
        trav = trav.left;
    }
    if(!s.isEmpty()) {
        trav = s.pop();
        print(trav.data);
        trav = trav.right;
    }
}
```

BST - BFS - Breadth First Search - Level wise Search - Non Rec



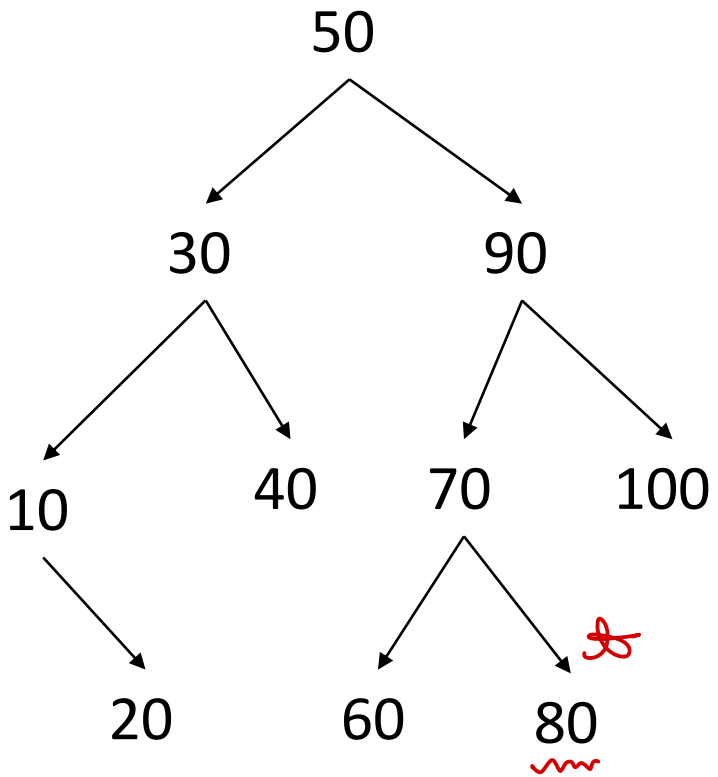
50 30 90 10 40
70 100 20 60 80

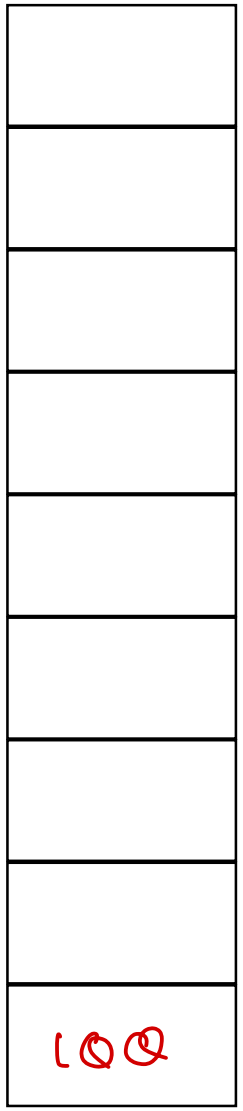
O(n)




```
q.push(root);
while(!q.isEmpty()) {
    trav = q.pop()
    if(key == trav.data)
        return trav;
    if(trav.left != null)
        q.push(trav.left);
    if(trav.right != null)
        q.push(trav.right);
}
return null;
```

BST - DFS - Depth First Search - O(n)

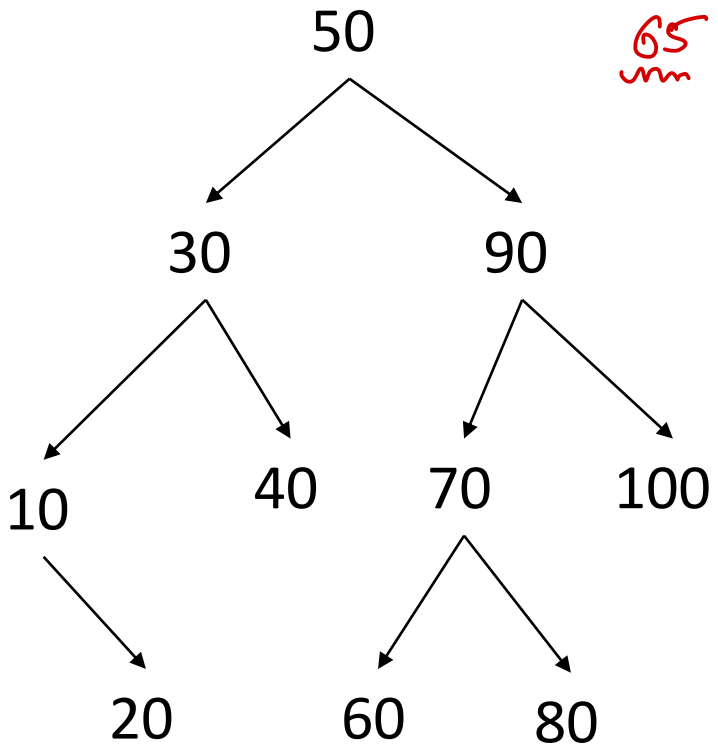




50 30 10 20 40 90 70 60 80

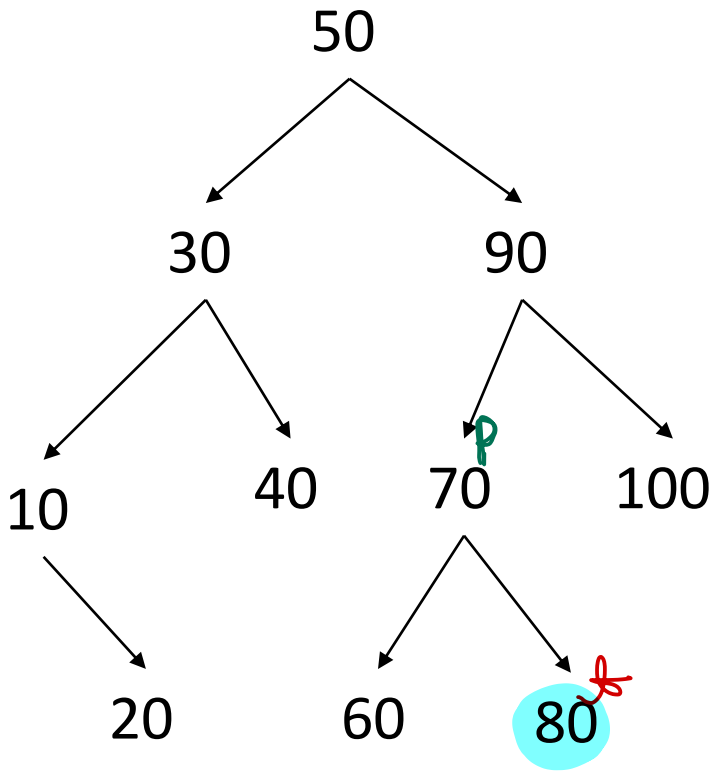
```
s.push(root);
while(!s.isEmpty()) {
    trav = s.pop();
    if(key == trav.data)
        return trav;
    if(trav.right != null)
        s.push(trav.right);
    if(trav.left != null)
        s.push(trav.left);
}
return null;
```

BST - Binary Search - O(h) → height



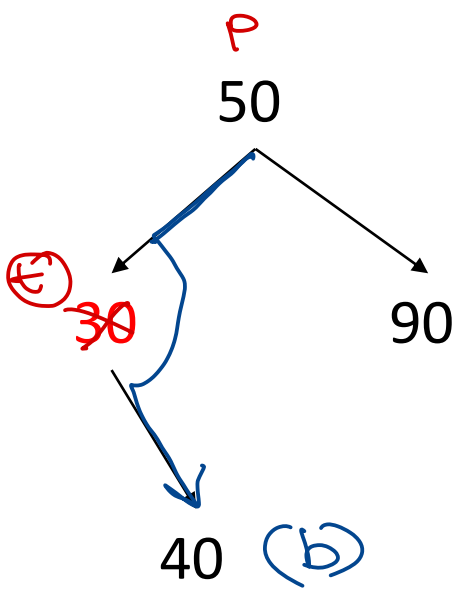

```
trav = root;
while(trav != null) {
    if(key == trav.data)
        return trav;
    if(key < trav.data)
        trav = trav.left;
    else
        trav = trav.right;
}
return null;
```

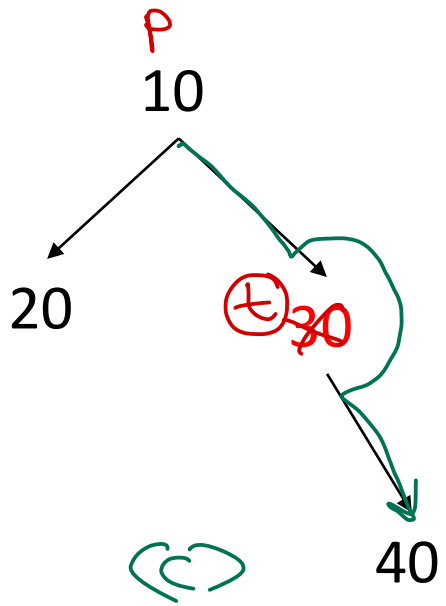
BST - Binary Search - O(h) → height

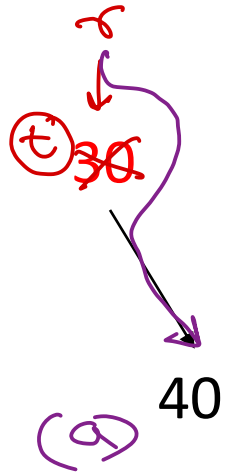



```
Parent = null;
trav = root;

while(trav != null) {
    if(key == trav.data)
        return {trav, parent};
    Parent = trav;
    if(key < trav.data)
        trav = trav.left;
    else
        trav = trav.right;
}

return {null, null};
```

BST – Delete Node → trav.left == null



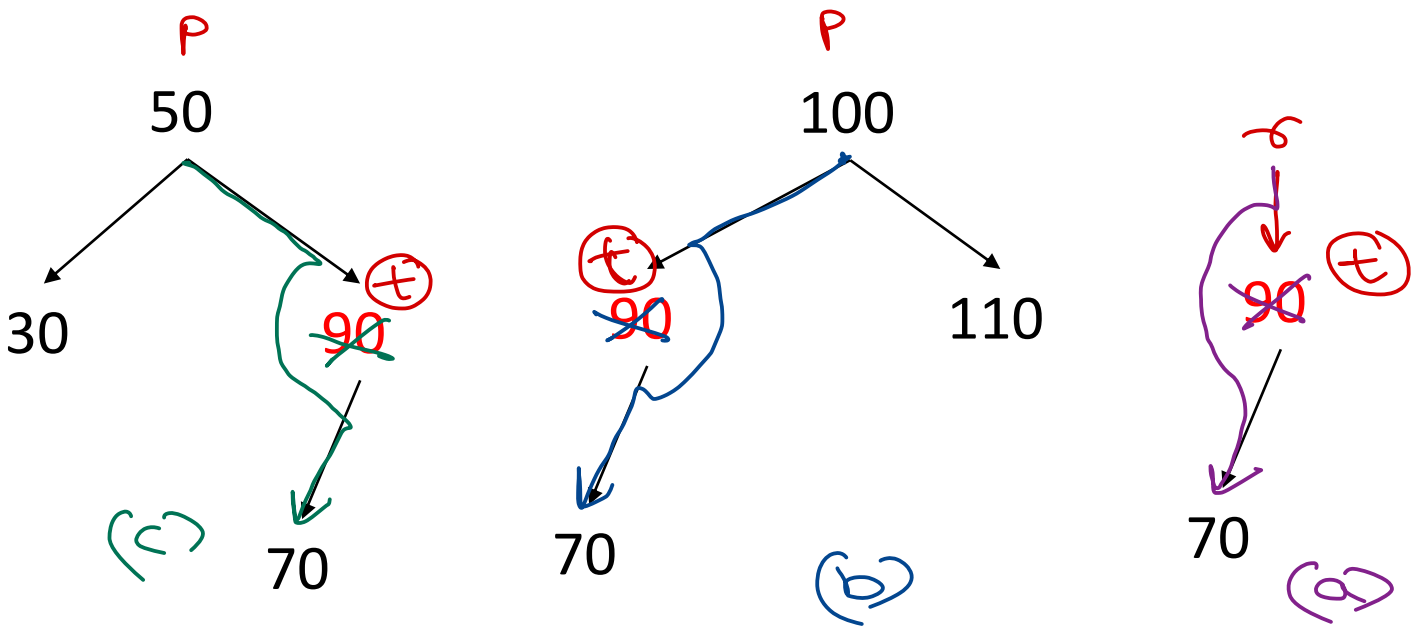





```
if(trav == root)
    root = trav.right;
else if(trav == p.left)
    p.left = trav.right;
else
    p.right = trav.right;
```

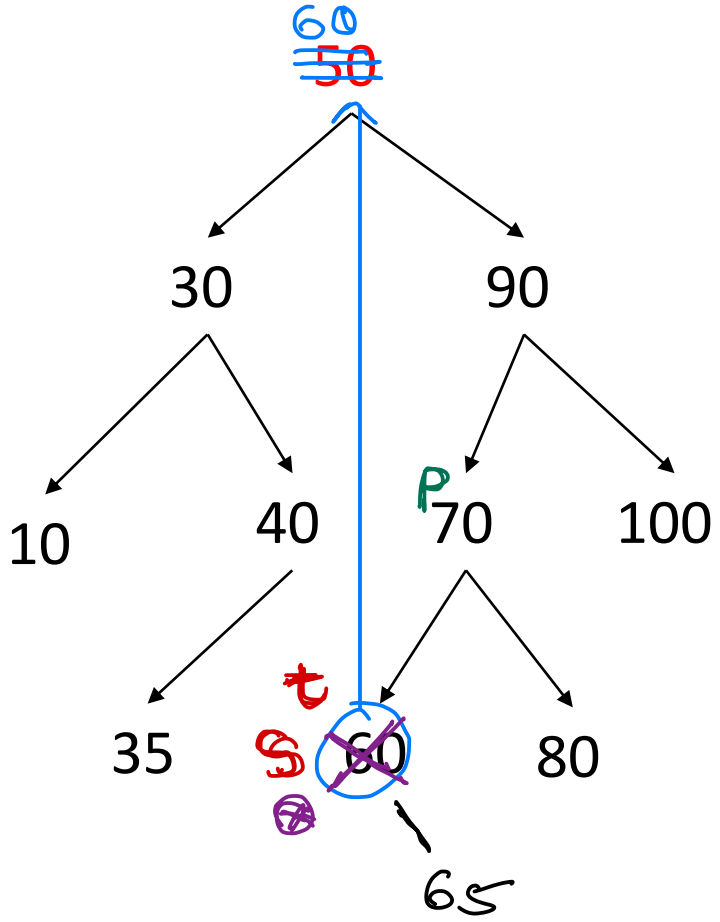
BST – Delete Node

trav->right == null



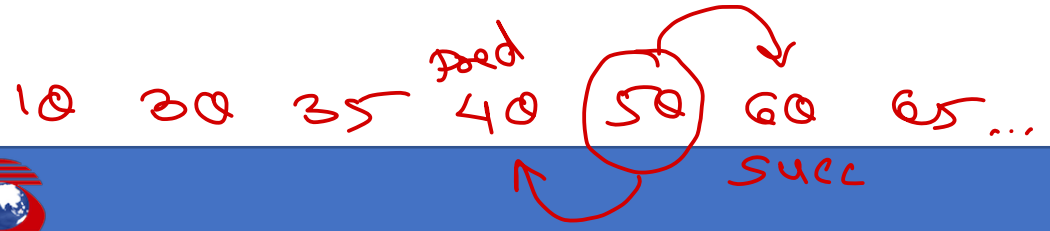

```
if(trav == root)
    root = trav->left;
else if(trav == p->left)
    p->left = trav->left;
else
    p->right = trav->left;
```

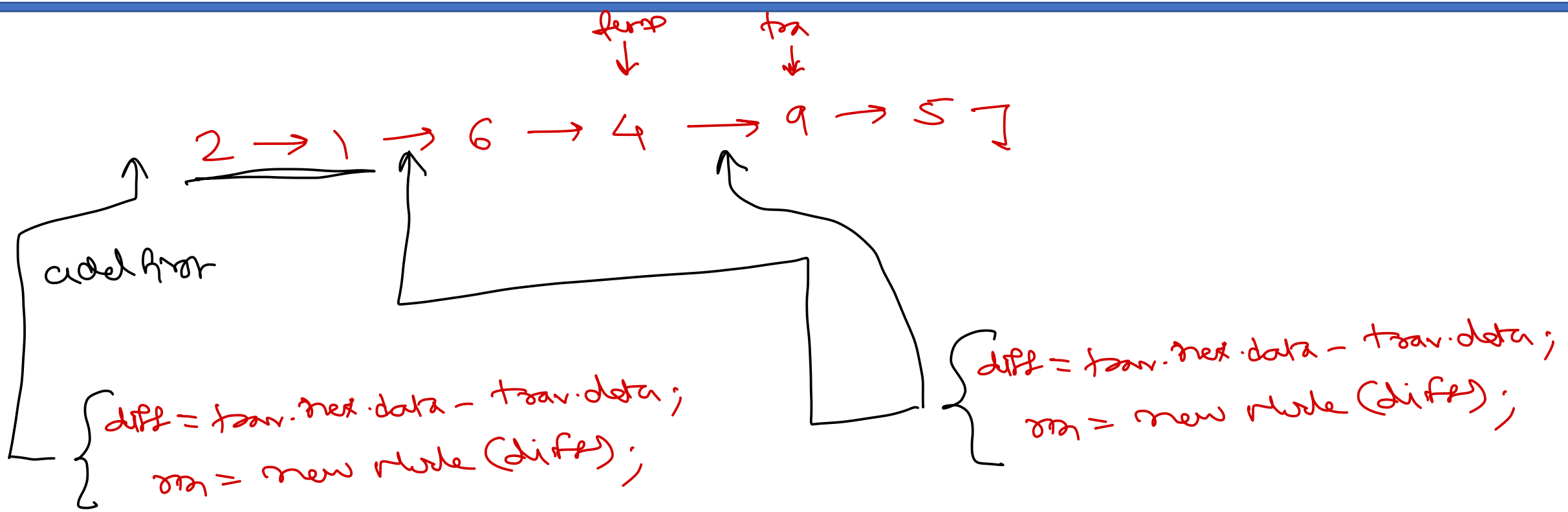
BST – Delete Node → trav.left != null && trav.right != null




```
parent = trav;
succ = trav.right;
while(succ.left != null){
    parent = succ;
    succ = succ.left;
}
trav.data = succ.data;
trav = succ;
```

```
parent.left = trav.right;
trav = null;
```




temp
trav
2 → 1 → 6 → 4 → 9 → 5
addNode
diff = trav.next.data - trav.data;
nn = new node(diff);
diff = trav.next.data - trav.data;
nn = new node(diff);

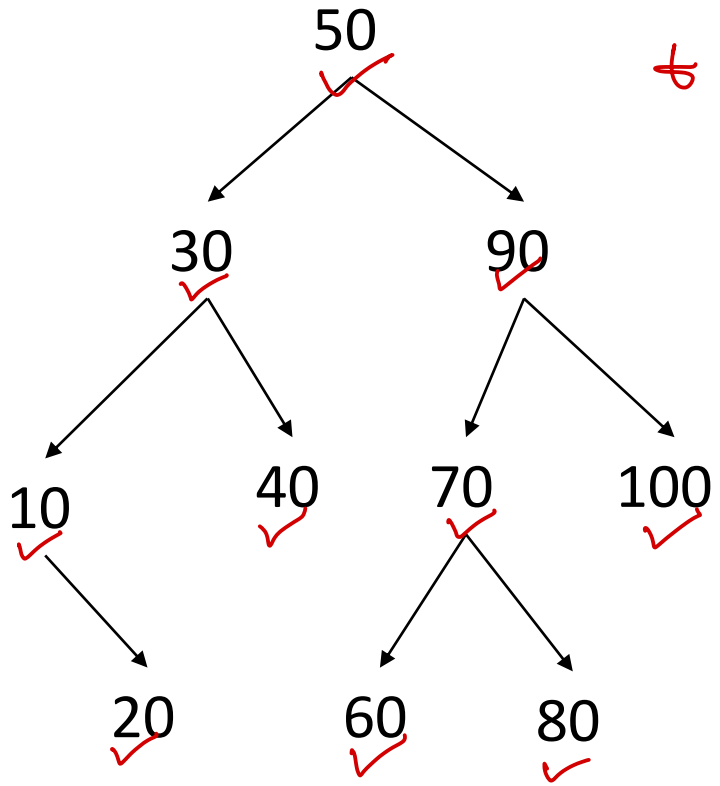
Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>

Data Structure & Algorithms

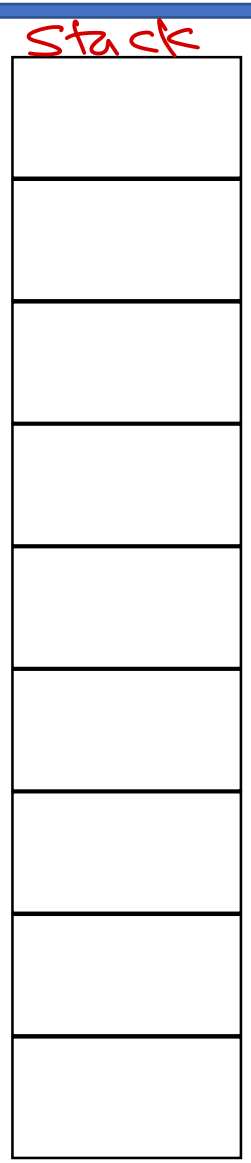
Sunbeam Infotech

BST - postorder()

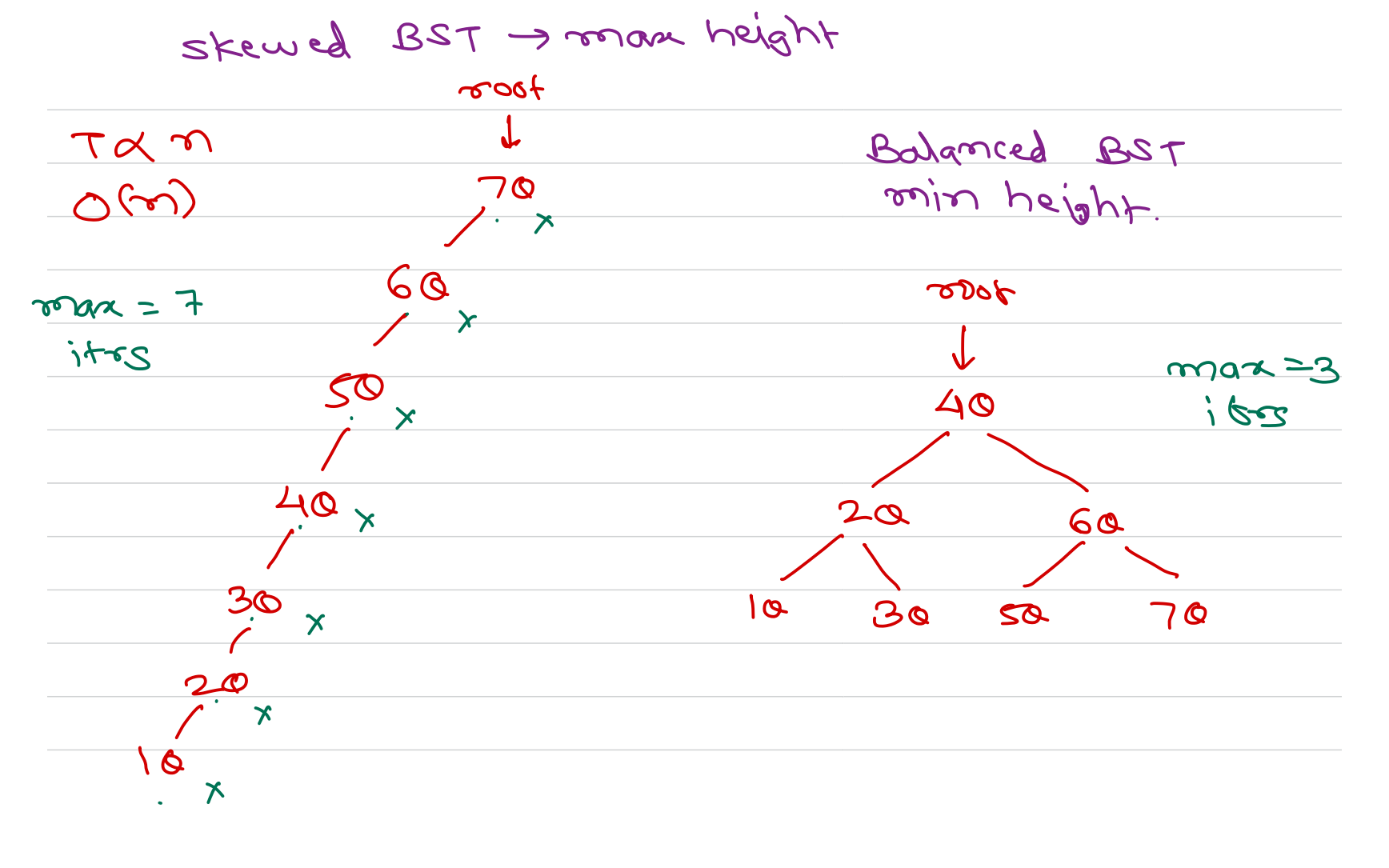


20 10 40 30 60

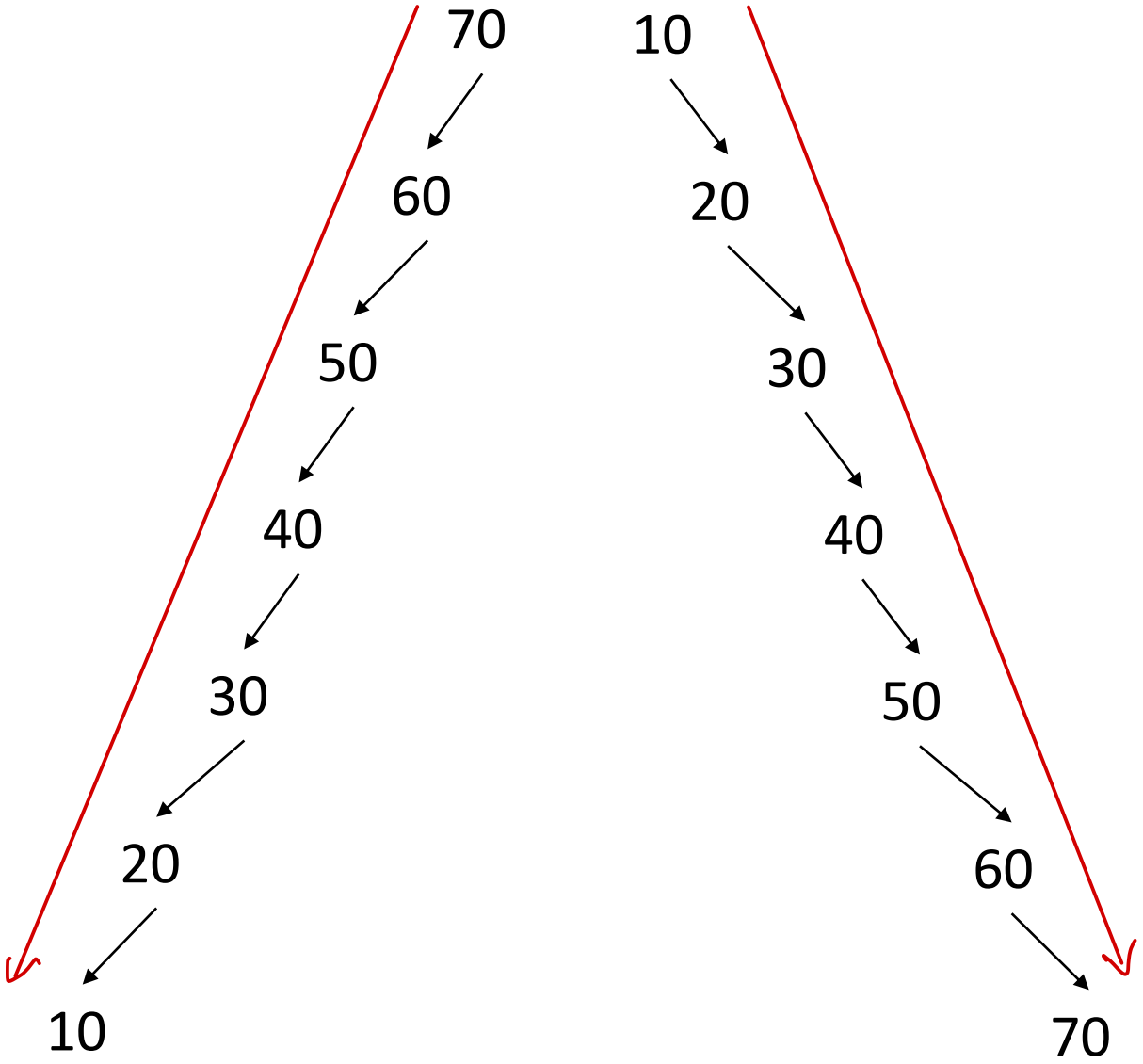
80 70 100 90 50




```
trav = root;
while(trav!=null || !s.isEmpty()){
    while(trav!=null){
        s.push(trav)
        trav = trav.left;
    }
    if(!s.isEmpty()){
        trav = s.pop();
        if(trav.right!=null &&
            !trav.right.visited){
            s.push(trav);
            trav = trav.right;
        }
        else{
            print(trav.data);
            trav.visited = true;
            trav=null;
        }
    }
}
```

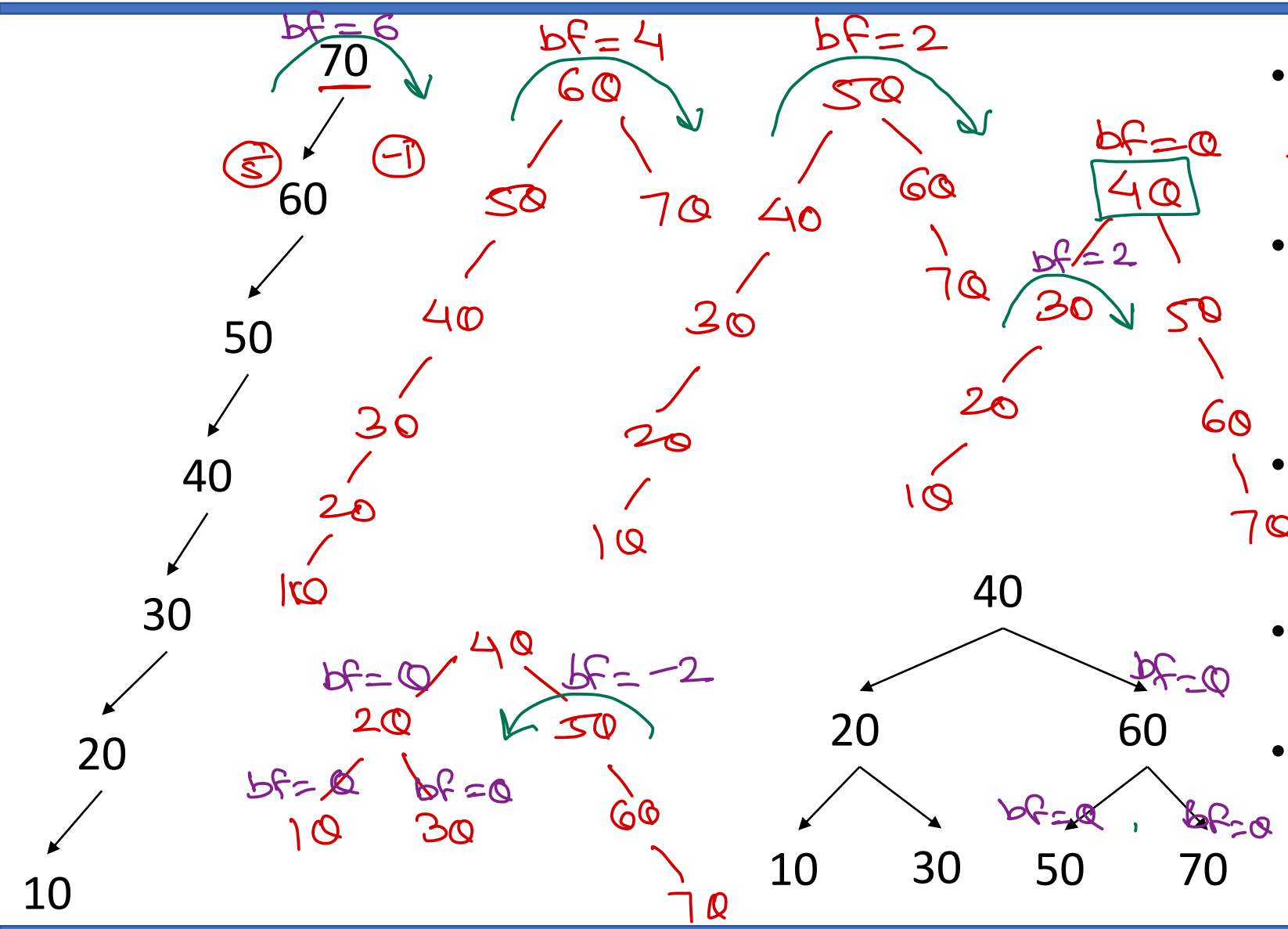

skewed BST → max height
root
70
60
50
40
30
20
10
T ∝ n
O(n)
max = 7
iters
Balanced BST
min height.
root
40
20
60
10
30
50
70
max = 3
iters

Skewed Binary Tree



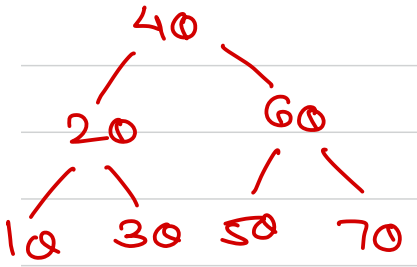
- In Binary tree if only left or only right links are used, tree grows only on one side. Such tree is called as skewed binary tree.
  - Left skewed binary tree
  - Right skewed binary tree
- Time complexity of any BST is O(h).
- Such tree have maximum height i.e. same as number of elements.
- Time complexity of searching in skewed BST is O(n).

Balanced BST



- To speed up searching, height of BST should minimum as possible.
- If nodes in BST are arranged so that its height is kept as less as possible, is called as Balanced BST.
- Balance factor
  - = Height of left sub tree – Height of ~~left~~ right sub tree
- In balanced BST, BF of each node is -1, 0 or +1.
- A tree can be balanced by applying series of left or right rotations on unbalanced nodes.

height = 2



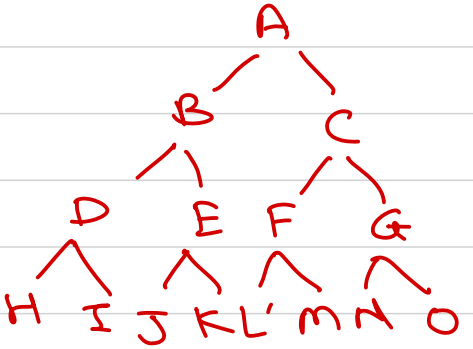
n = 7

$n = 2^{h+1} - 1$

leaf nodes (external) $= 2^h$

non-leaf nodes (internal) $= 2^h - 1$

height = 3

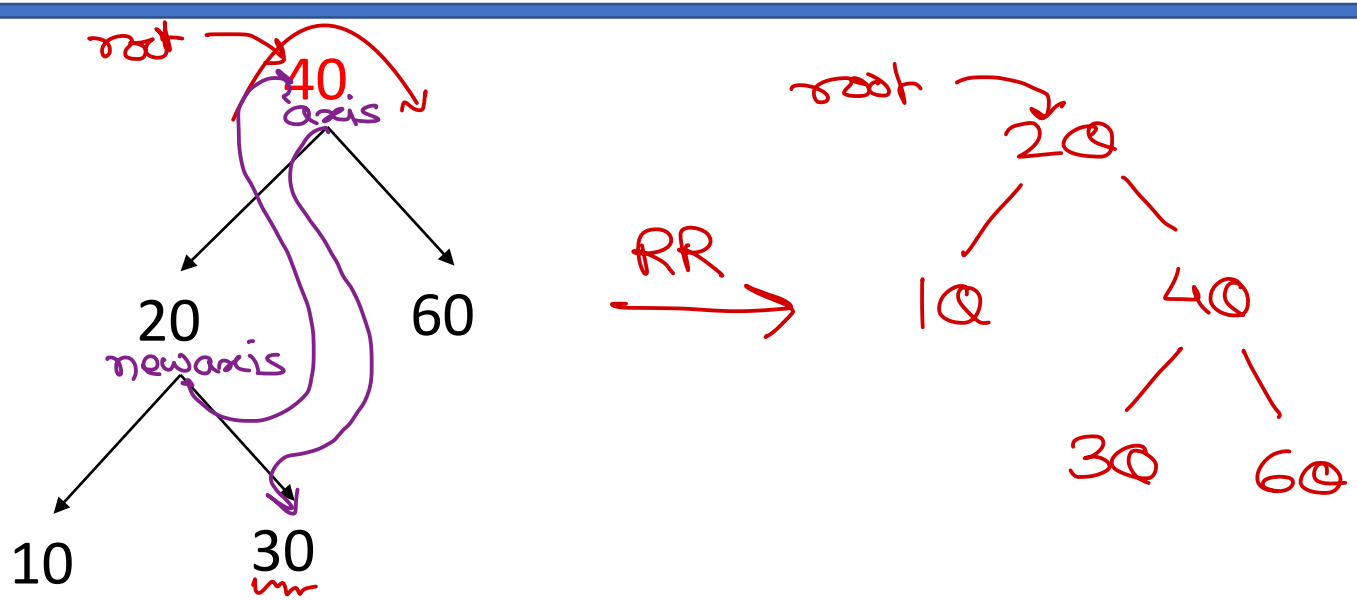


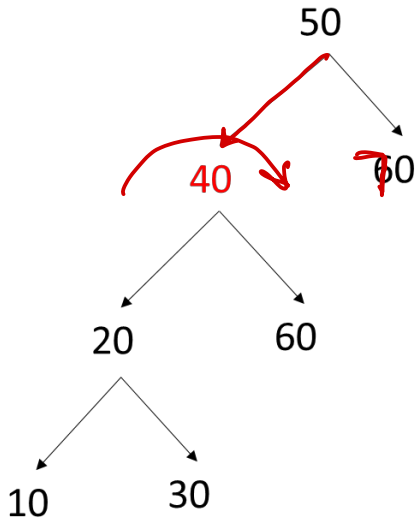
n = 15

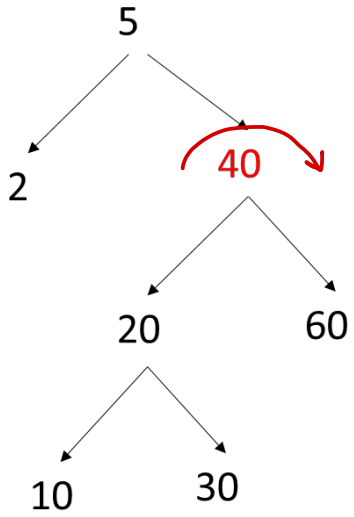
leaf = 8

non-leaf = 7

Right rotation

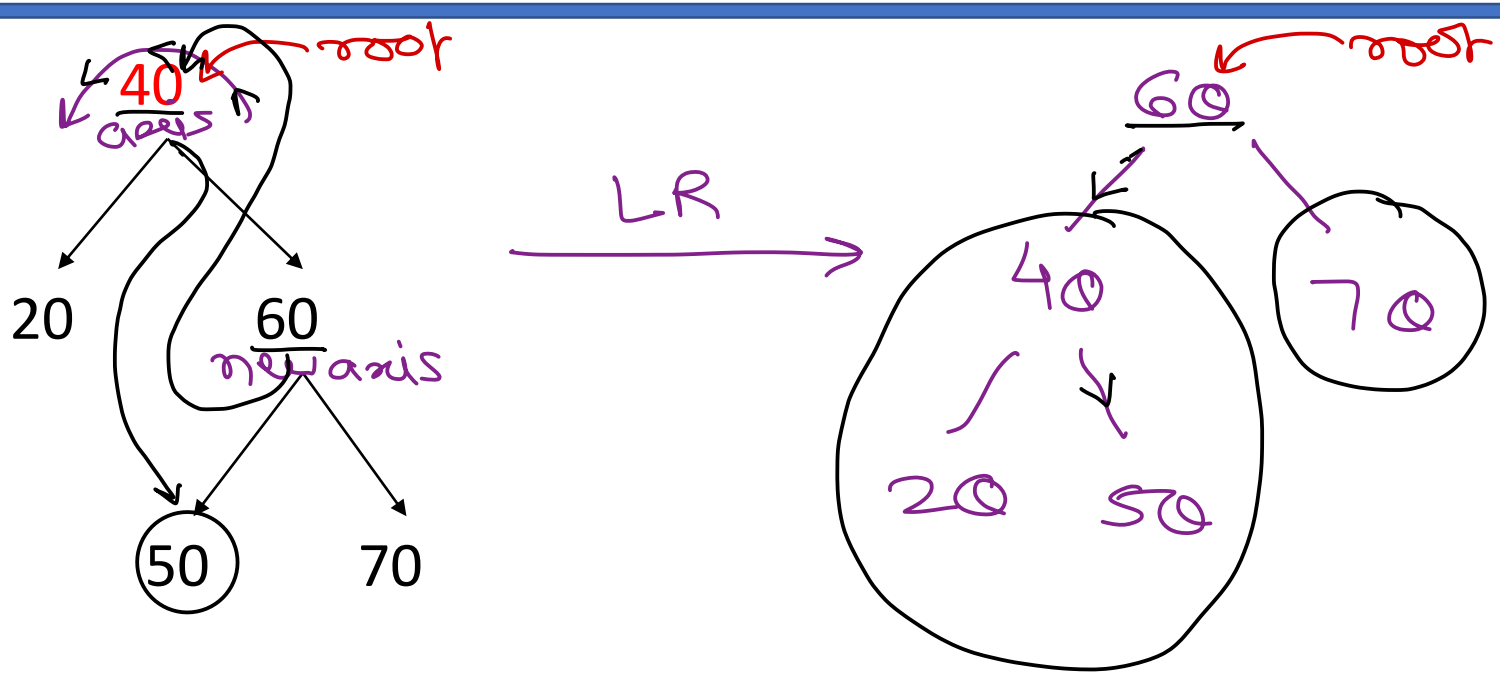


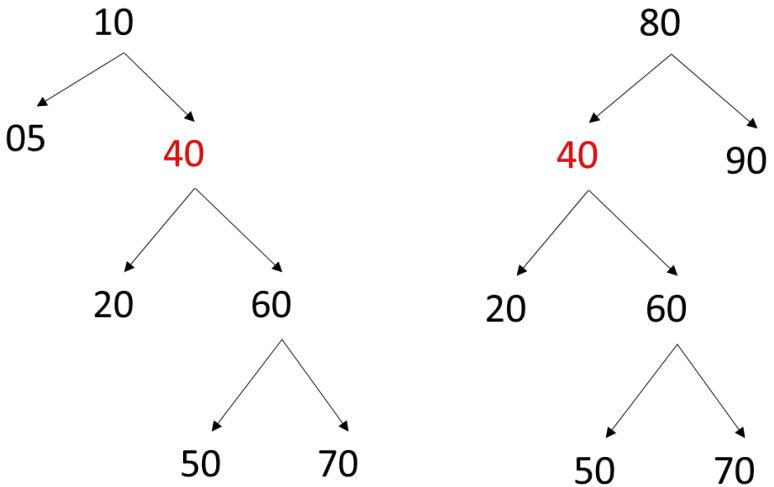


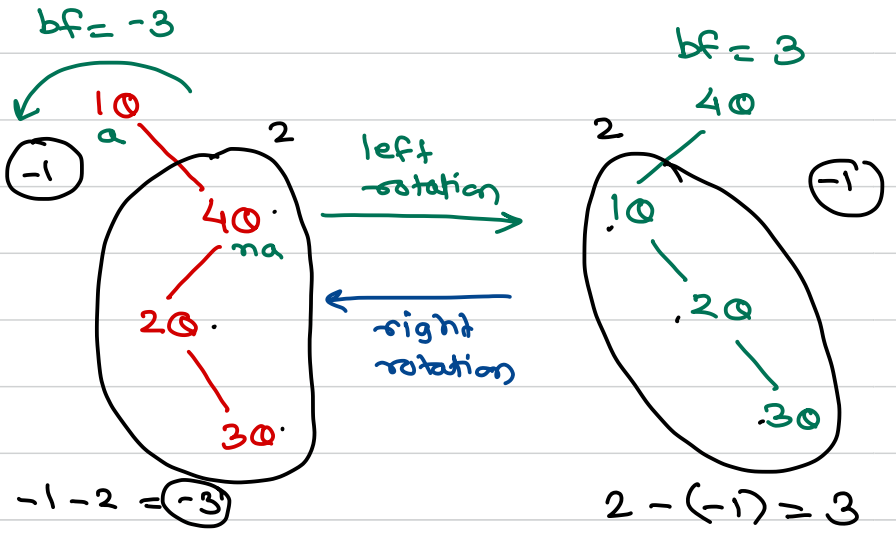



```
newaxis = axis.left;
axis.left = newaxis.right;
newaxis.right = axis;
if(axis == root)
    root = new axis;
```

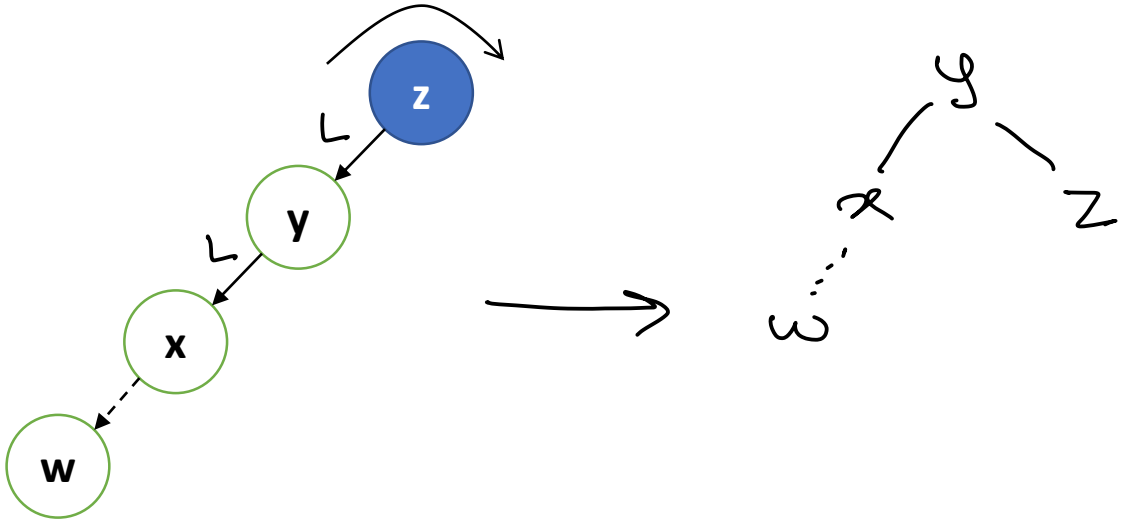
Left rotation





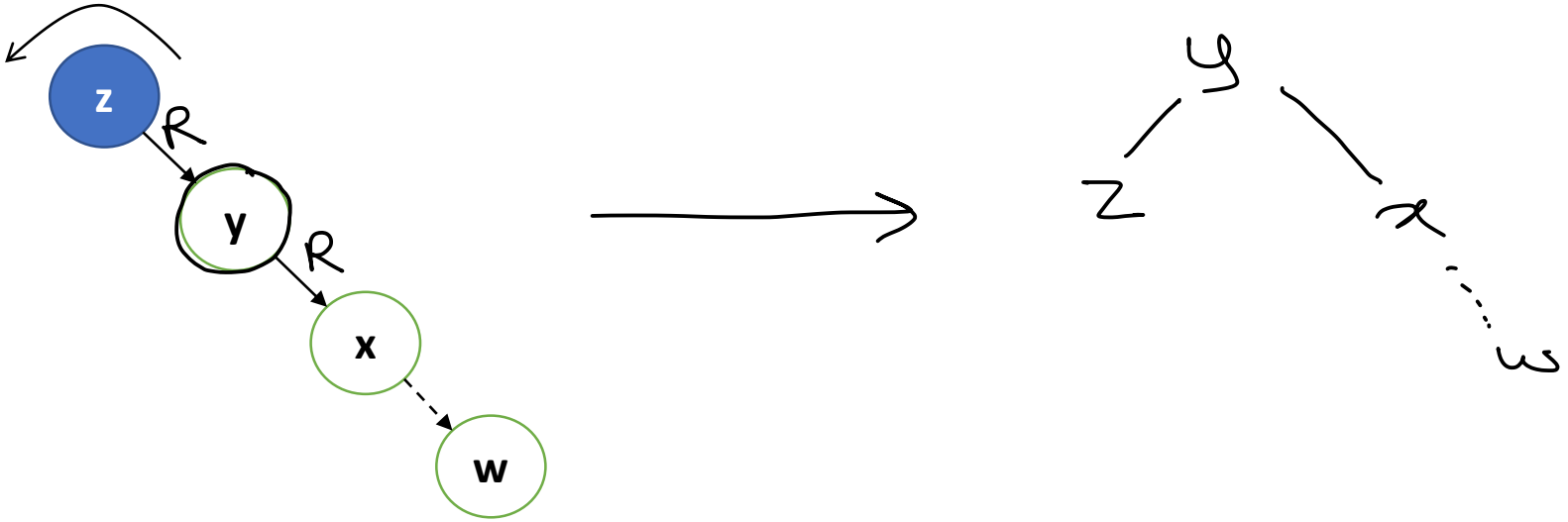

bf = -3
10
a
-1
2
40
na
20
30
-1 - 2 = -3
left
rotation
right
rotation
bf = 3
40
2
10
-1
20
30
2 - (-1) = 3

Rotation cases



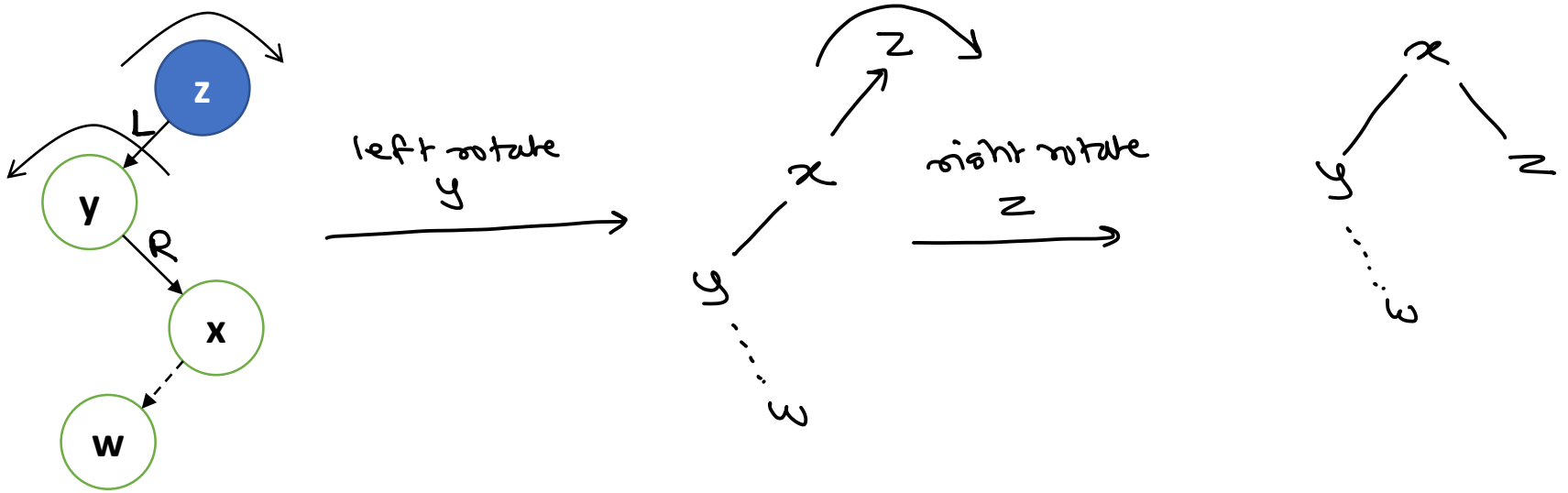
Left-Left case

Rotation cases



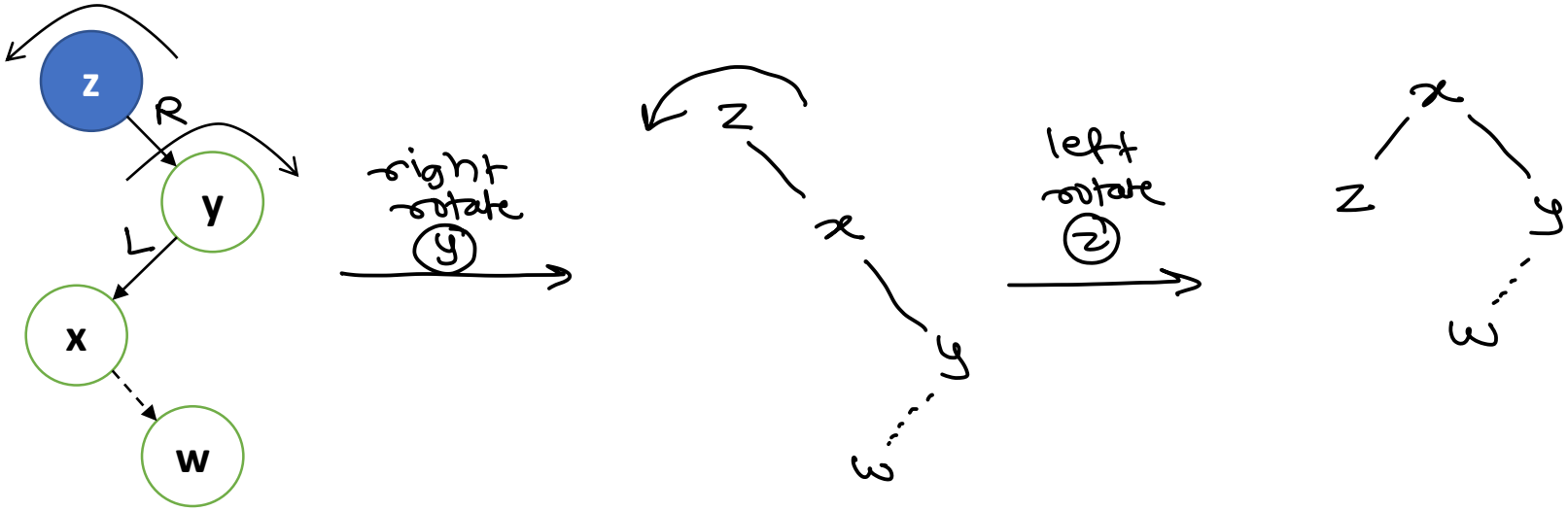
Right-Right case

Rotation cases

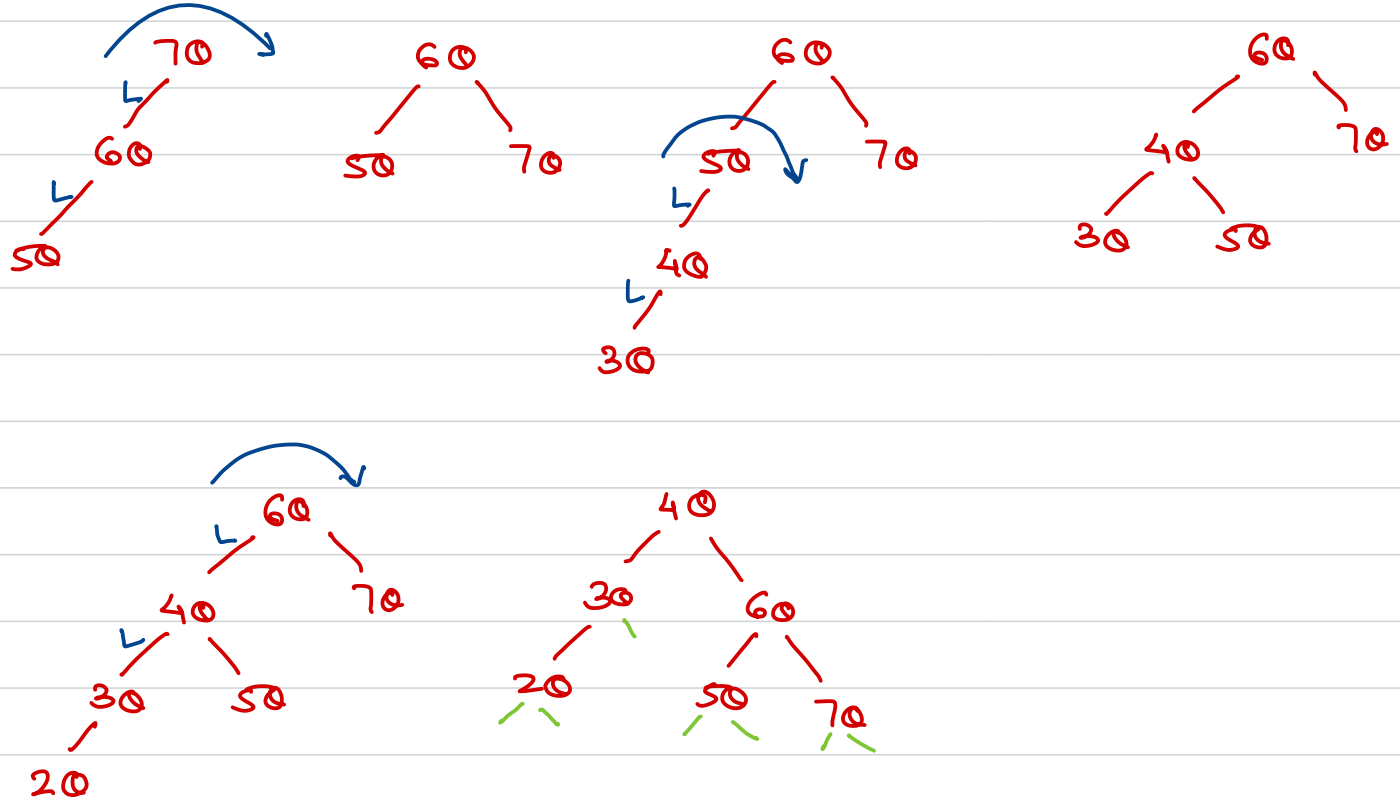


Left-Right case

Rotation cases



Right-Left case


70
60
50
60
50
70
60
50
70
40
30
60
40
70
30
50
60
40
70
30
50
20
40
30
60
20
50
70

AVL Tree

- AVL tree is a self-balancing Binary Search Tree (BST).
- The difference between heights of left and right subtrees cannot be more than one for all nodes.
- Most of BST operations are done in O(h) i.e. O(log n) time.
- Nodes are rebalanced on each insert operation and delete operation.
- Need more number of rotations as compared to Red & Black tree.

$$2^h \simeq n$$

$$h \log 2 = \log n$$

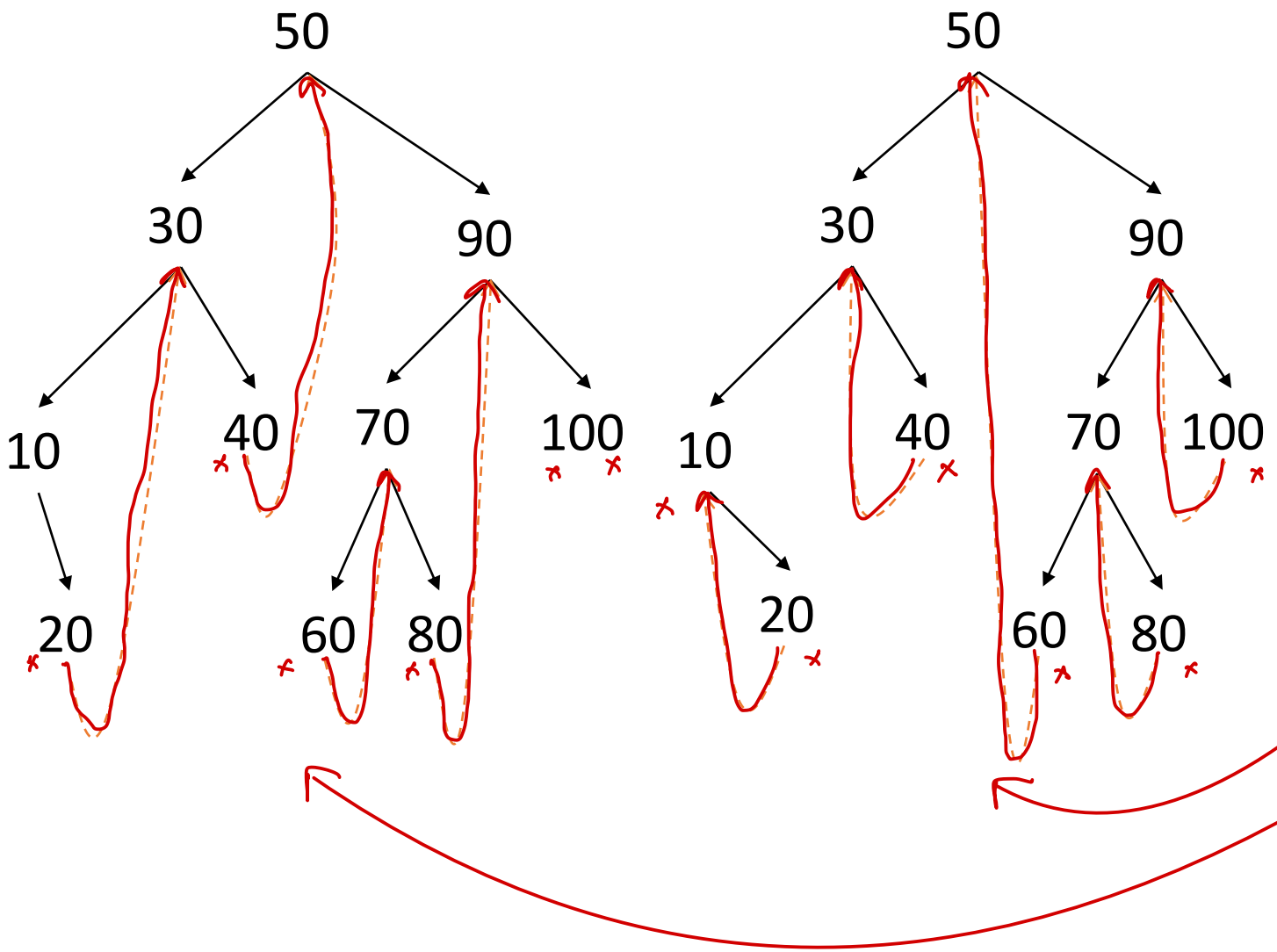
$$h = \frac{\log n}{\log 2}$$

$$T \propto h$$

$$T \propto \frac{\log n}{\log 2}$$

$$O(\log n)$$

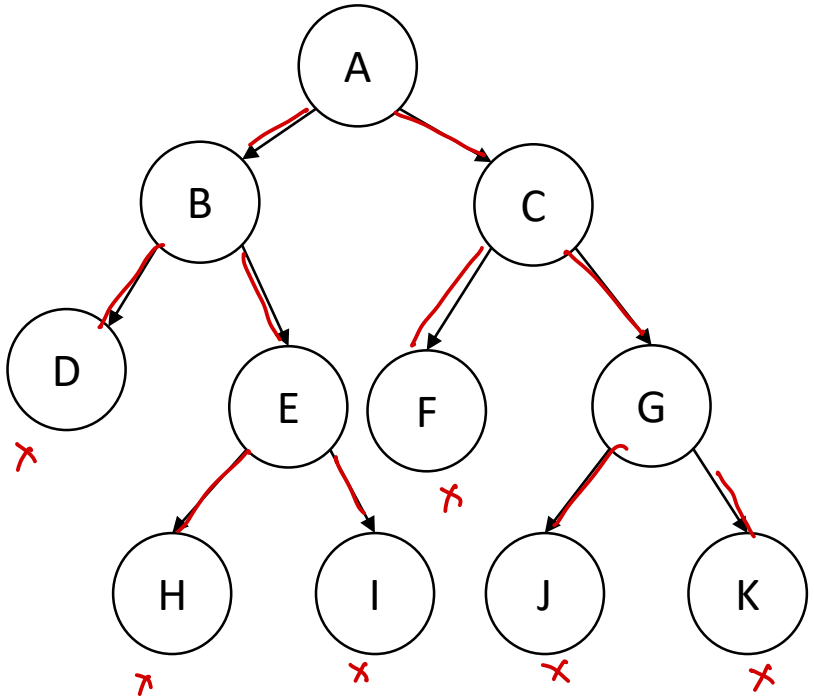
Threaded BST



- Typical BST in-order traversal involves recursion or stack. It slows execution and also need more space.
- Threaded BST keep address of in-order successor or predecessor addresses instead of NULL to speed up in-order traversal (using a loop).
- Left threaded BST
- Right threaded BST
- In-threaded BST

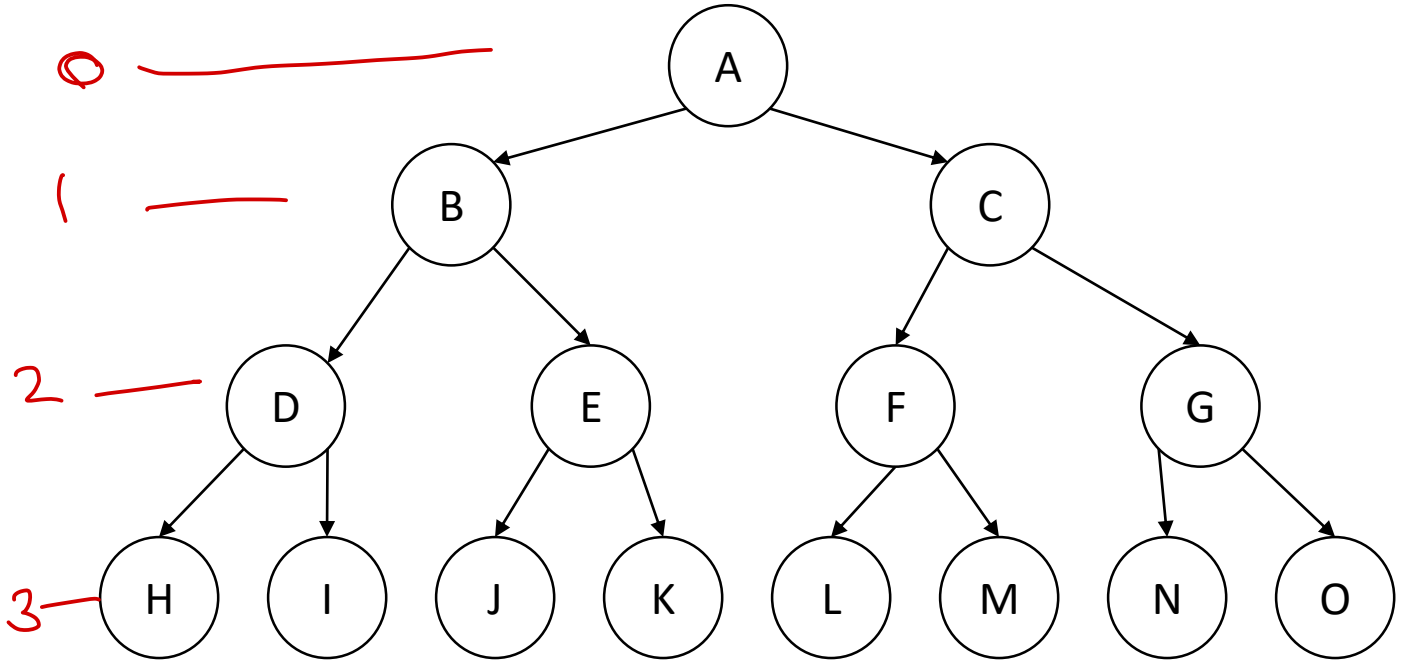
left + right

Strict Binary Tree

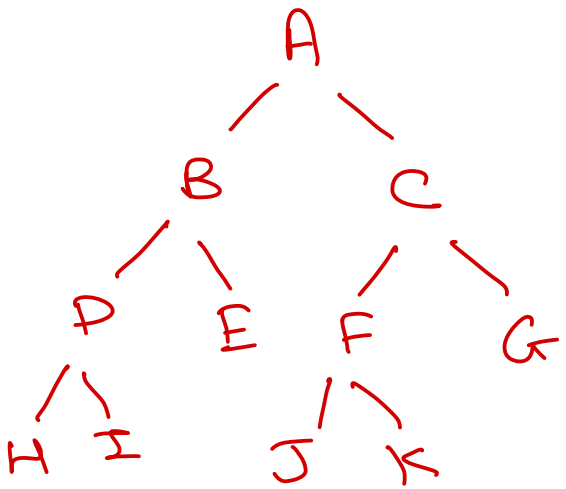


- Binary tree in which each non-leaf node has exactly two child nodes.

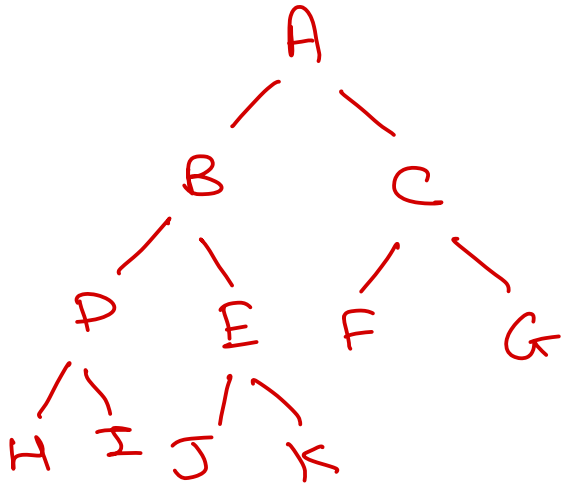
Perfect Binary Tree



- Binary tree which is full for the given height i.e. contains maximum possible nodes.
- Number of nodes = $2^h - 1$ $2^{h+1} - 1$

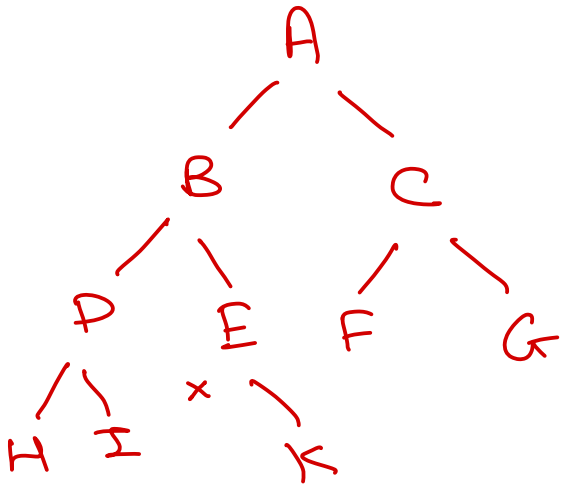


Strict



Strict

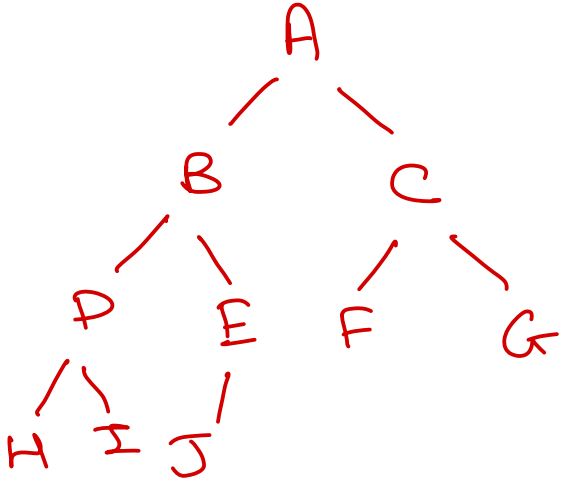
Complete



Strict X

Complete X

almost Complete X

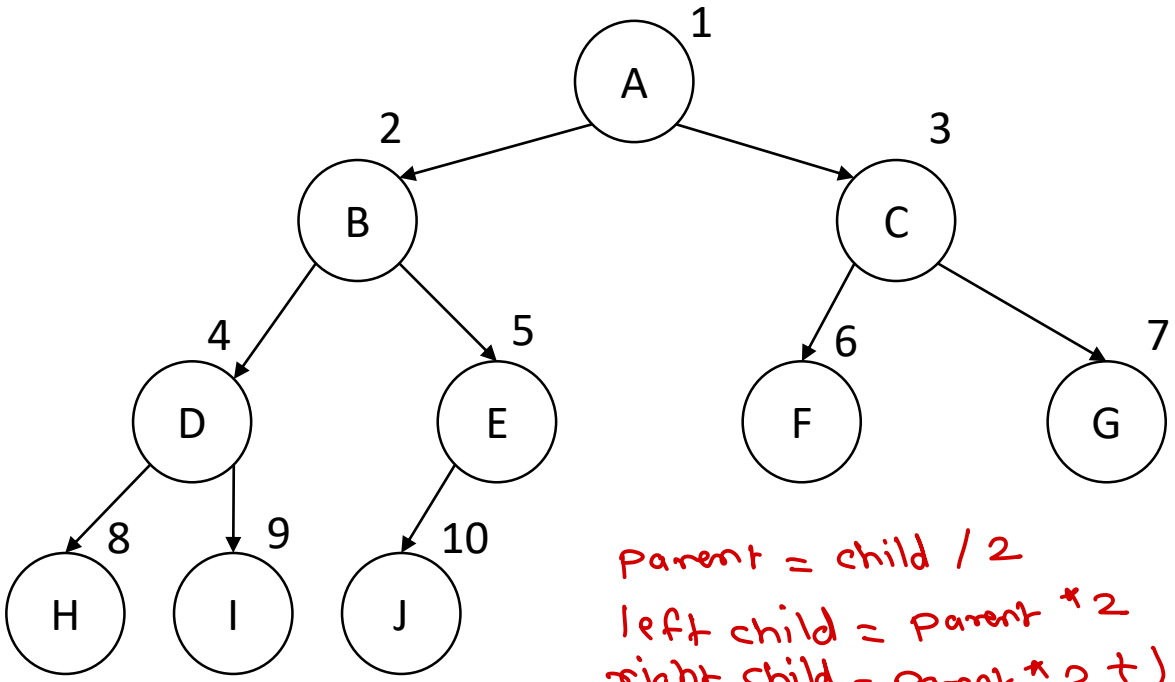


Strict X

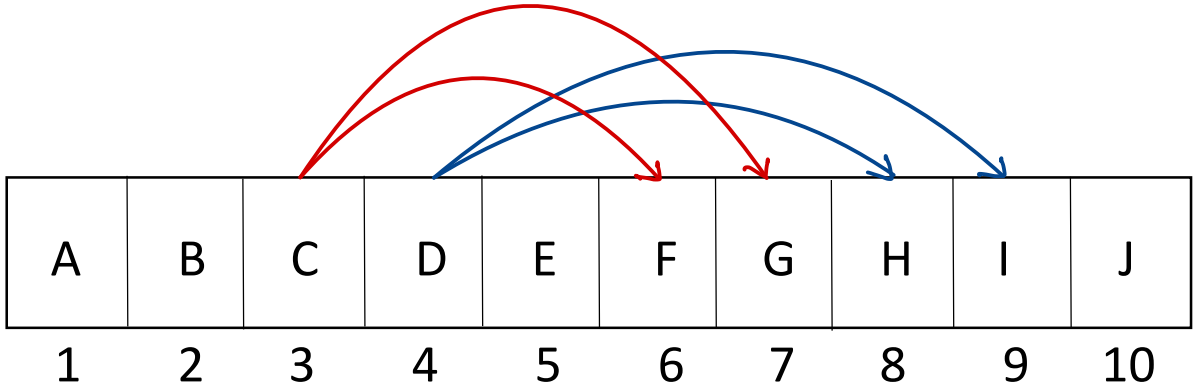
complete ✓

✓ almost complete

Complete Binary Tree and Heap



parent = child / 2
left child = parent * 2
right child = parent * 2 + 1

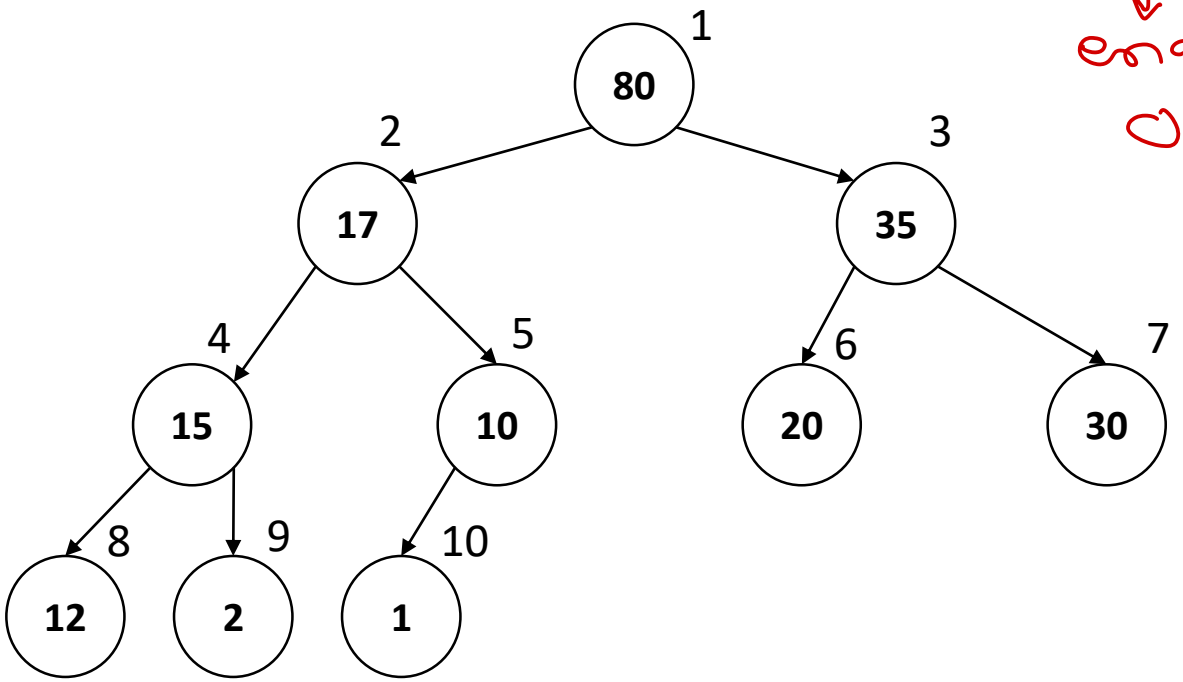


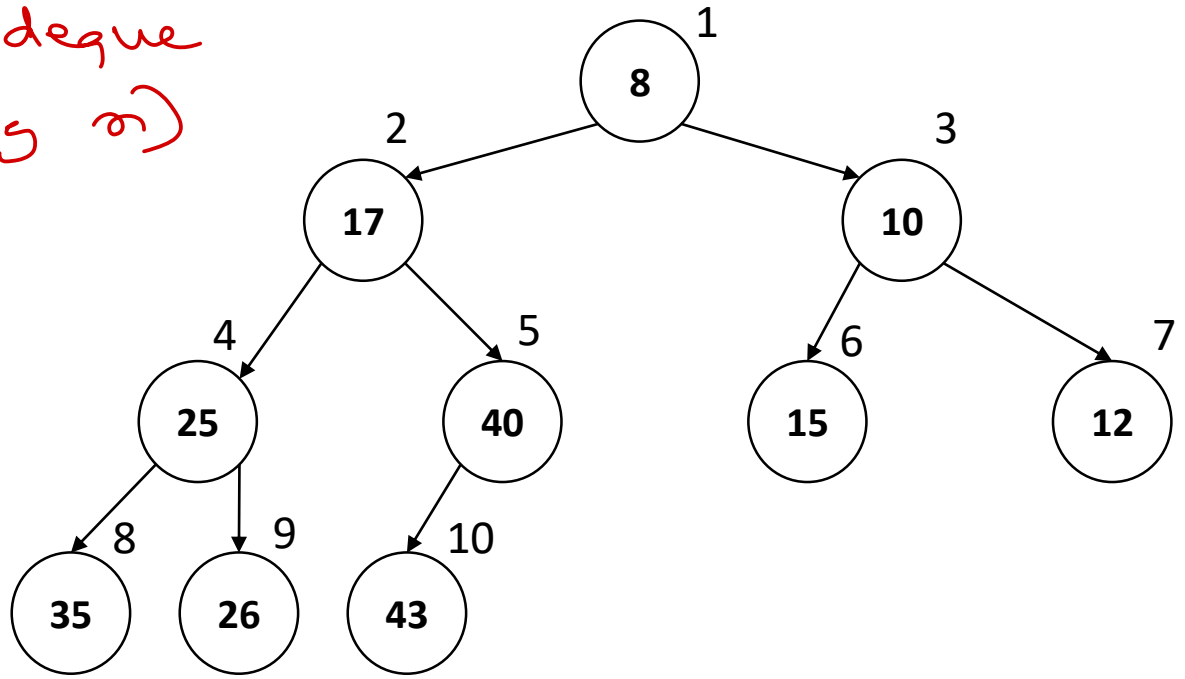
- A complete binary tree (CBT) is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- Almost complete binary tree is similar to CBT, but may not be strictly binary tree.

- Heap is array implementation of complete binary tree.
- Parent child relation is maintained through index calculations
  - parent index = child index / 2
  - left child index = parent index * 2
  - right child index = parent index * 2 + 1

Max Heap & Min Heap → used to implement priority queues.

enque/deque
O(log n)

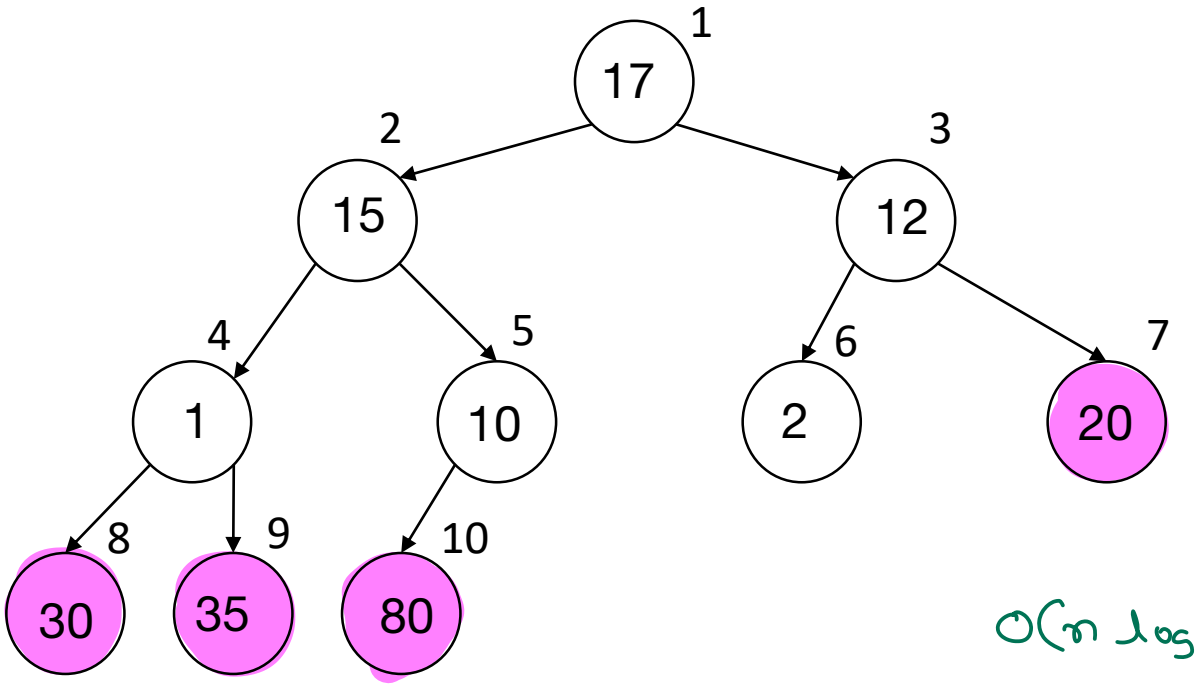




- Max heap is a heap data structure in which each node is greater than both of its child nodes.

- ~~Max~~ min heap is a heap data structure in which each node is smaller than both of its child nodes.

Heap Sort

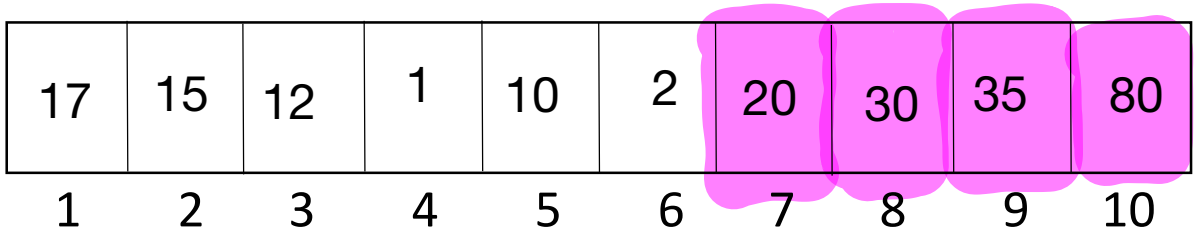


Heap Sort → O(n log n)

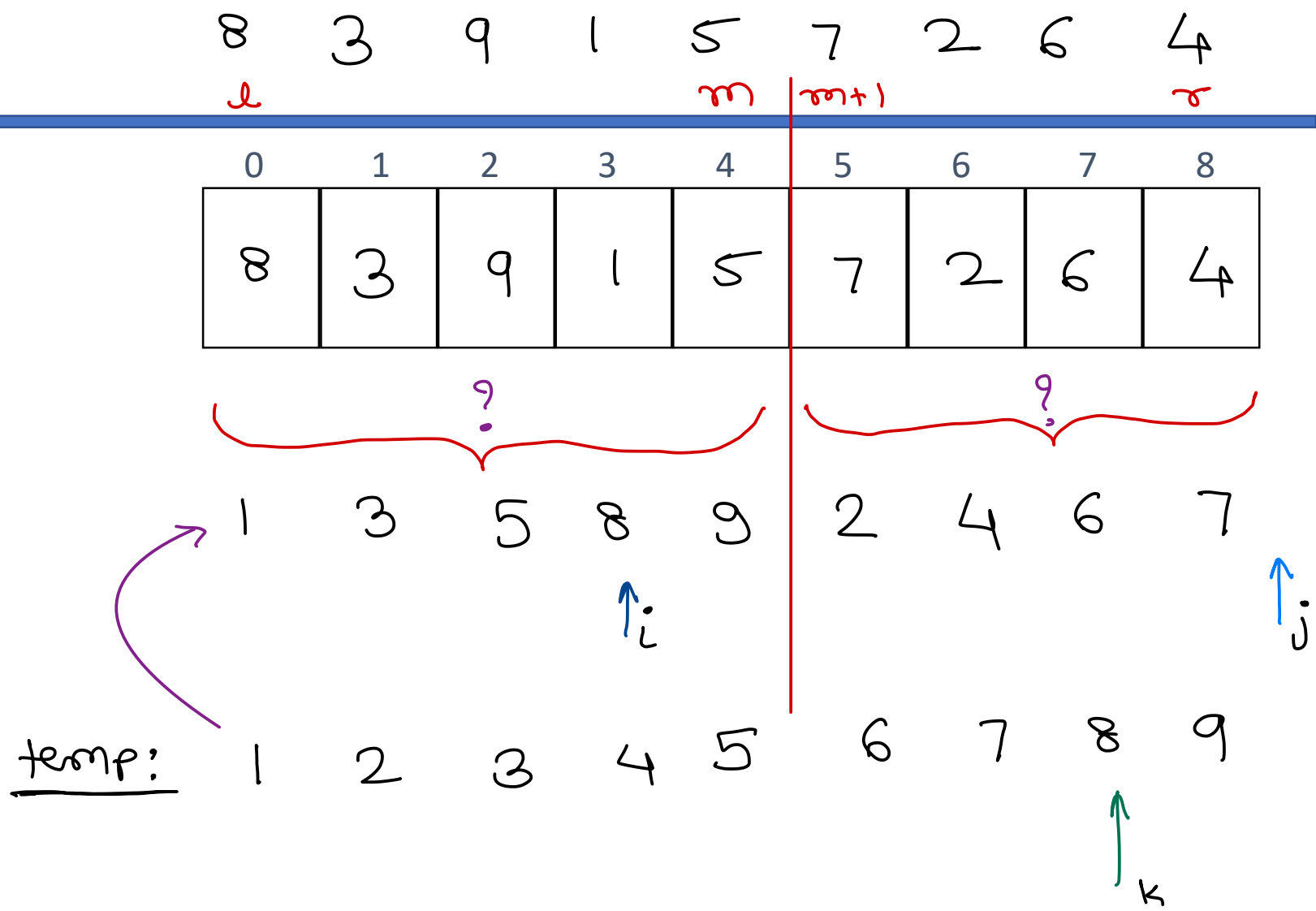
① make max heap; → O(n log n)

② while (heap is not empty)
{
del ele from heap; → O(log n)
put val at end;
}

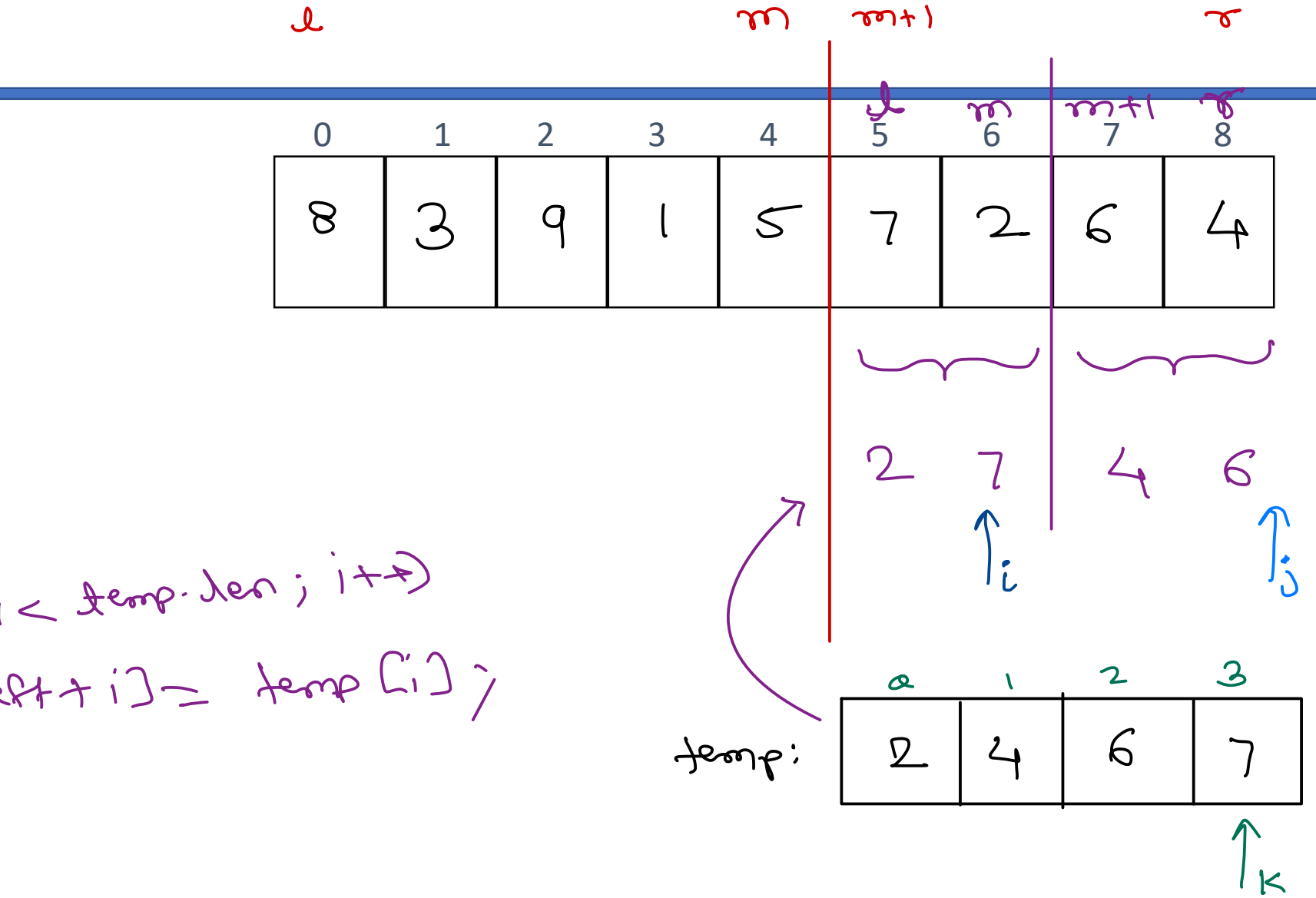
O(n log n)



Merge Sort


8 3 9 1 5 7 2 6 4
l
m
m+1
r
0 1 2 3 4 5 6 7 8
8 3 9 1 5 7 2 6 4
?
?
1 3 5 8 9 2 4 6 7
i
j
temp:
1 2 3 4 5 6 7 8 9
k

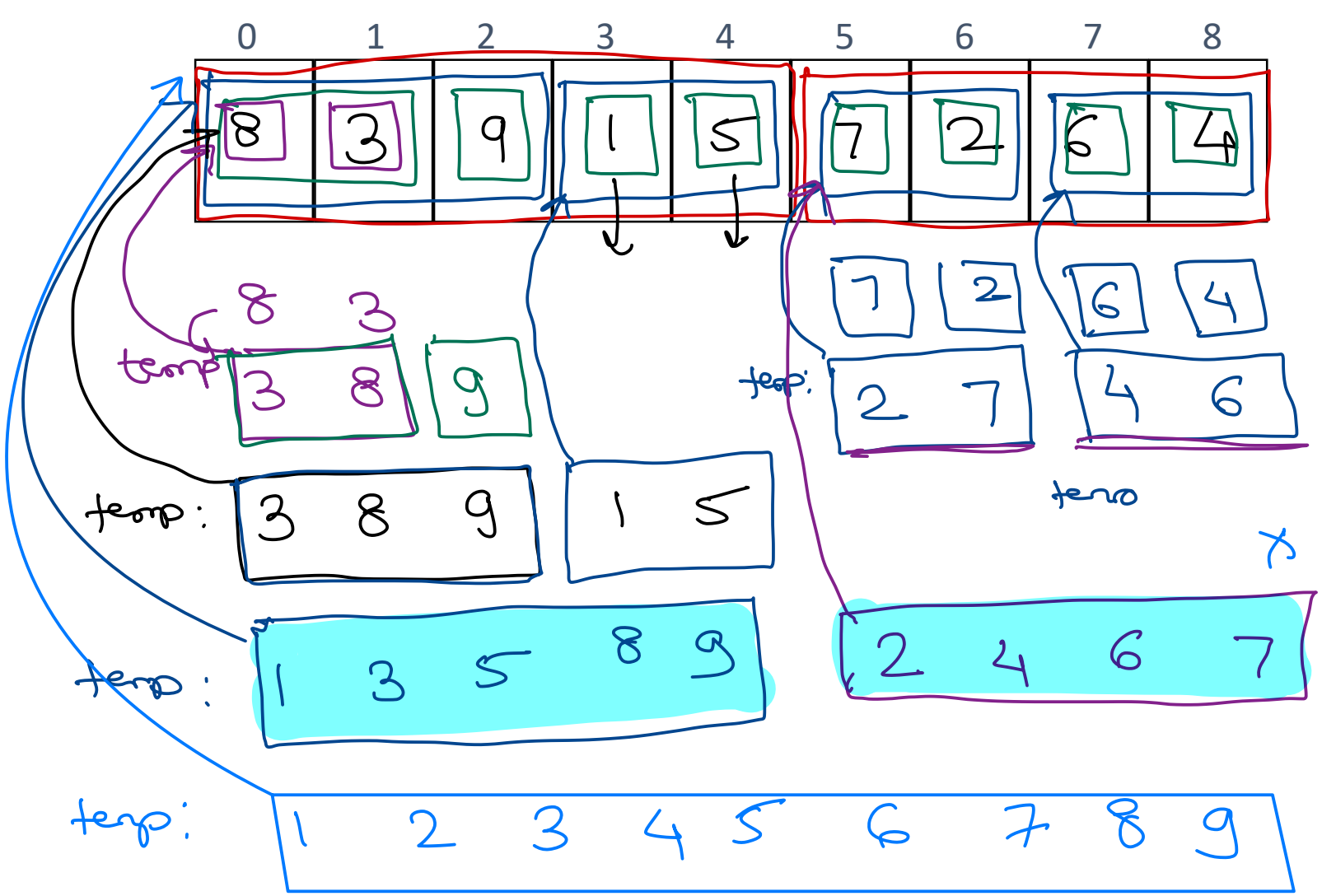
Merge Sort




```
for (i=0; i< temp.len ; i++)
    arr[left+i] = temp[i];
```

Merge Sort

l=0 m=4 | 5 r=8



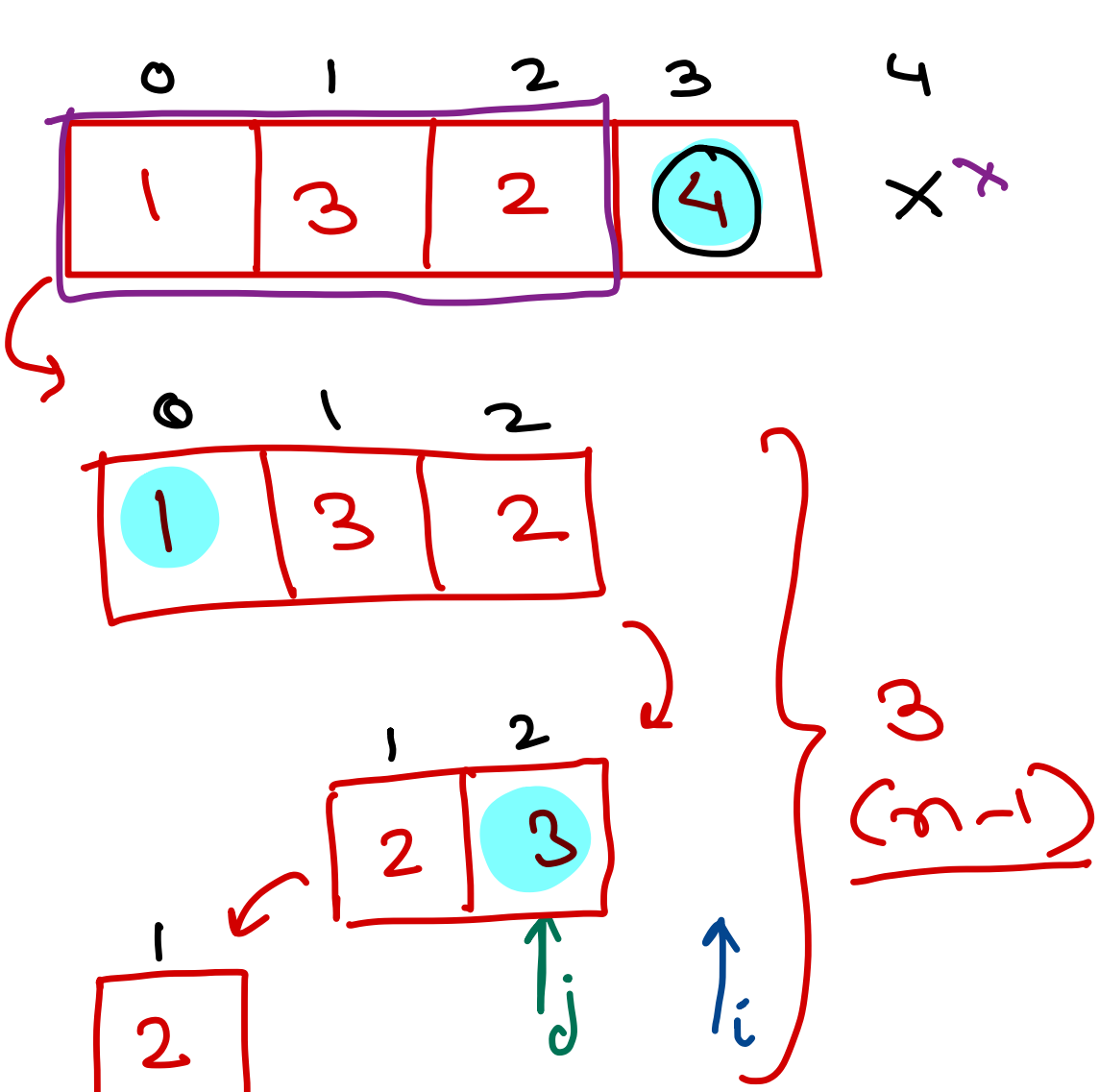
Thank you!

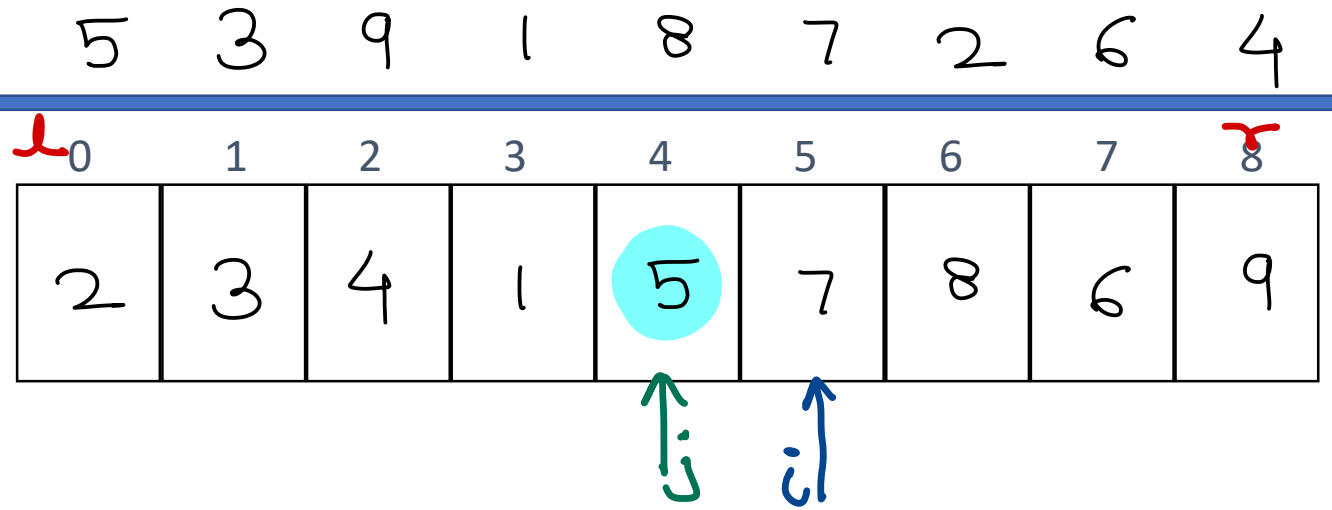
Nilesh Ghule <nilesh@sunbeaminfo.com>

Data Structure & Algorithms

Sunbeam Infotech

Quick Sort


0 1 2 3
1 3 2 4
4
x x
0 1 2
1 3 2
1 2
2 3
j
i
1
2
3
(n-1)


5 3 9 1 8 7 2 6 4
l
r
0 1 2 3 4 5 6 7 8
2 3 4 1 5 7 8 6 9
j
i

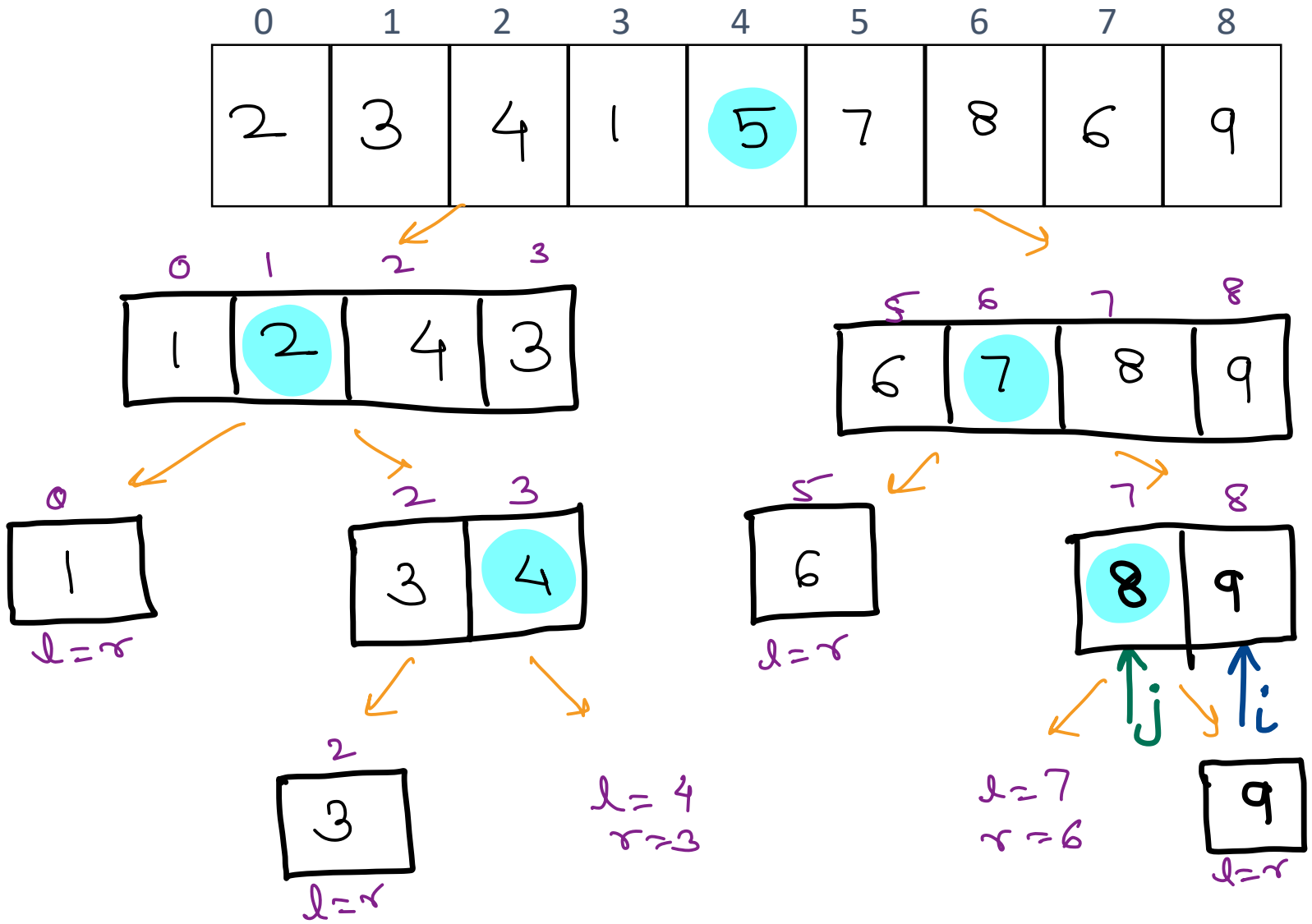
Quick Sort

Quick Sort

↳ O(n log n)

worst case

↳ O(n²)



Quick Sort – Time complexity

- Quick sort pivot element can be
  - First element or Last element
  - Random element
  - Median of the array → middle element in sorted array.
- Quick sort time
  - Time to partition as per pivot – T(n)
  - Time to sort left partition – T(k)
  - Time to sort left partition – T(n-k-1)
- Worst case
  - $T(n) = T(0) + T(n-1) + O(n) \Rightarrow O(n^2)$
- Best case
  - $T(n) = T(n/2) + T(n/2) + O(n) \Rightarrow O(n \log n)$
- Average case
  - $T(n) = T(n/9) + T(9n/10) + O(n) \Rightarrow O(n \log n)$

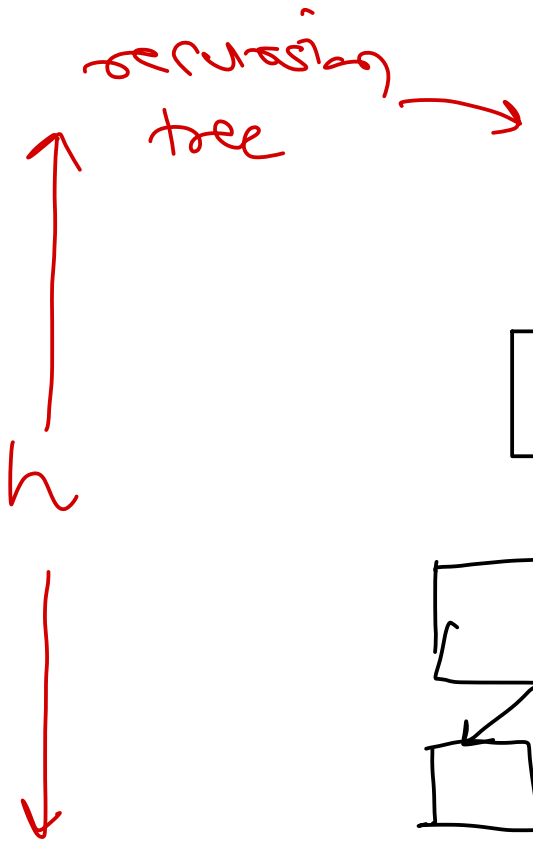
Merge Sort

$h \simeq \log n$

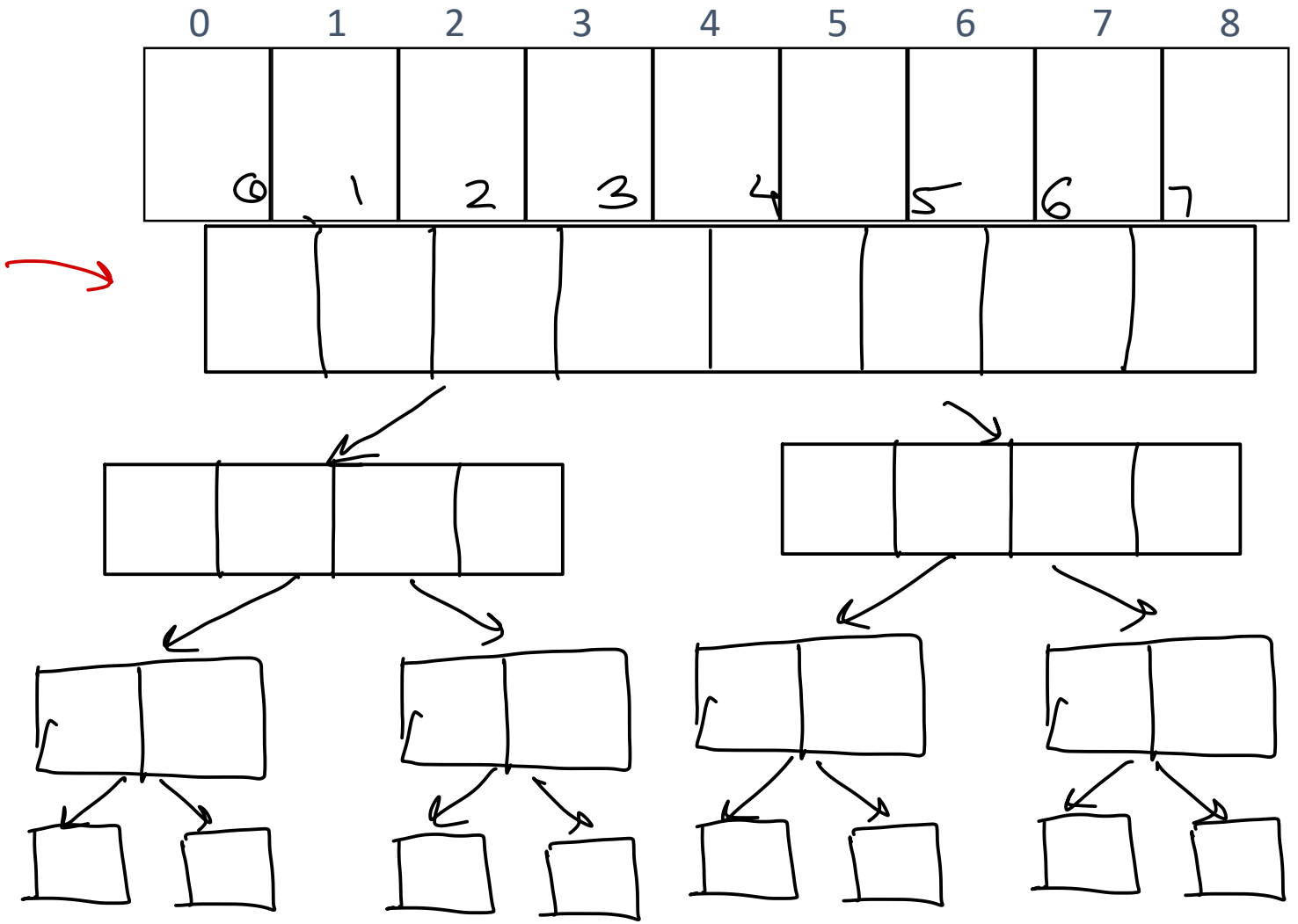
* partitioning
  ↳ $\log n$

* merging
  ↳ n

* time complexity
  $O(n \log n)$



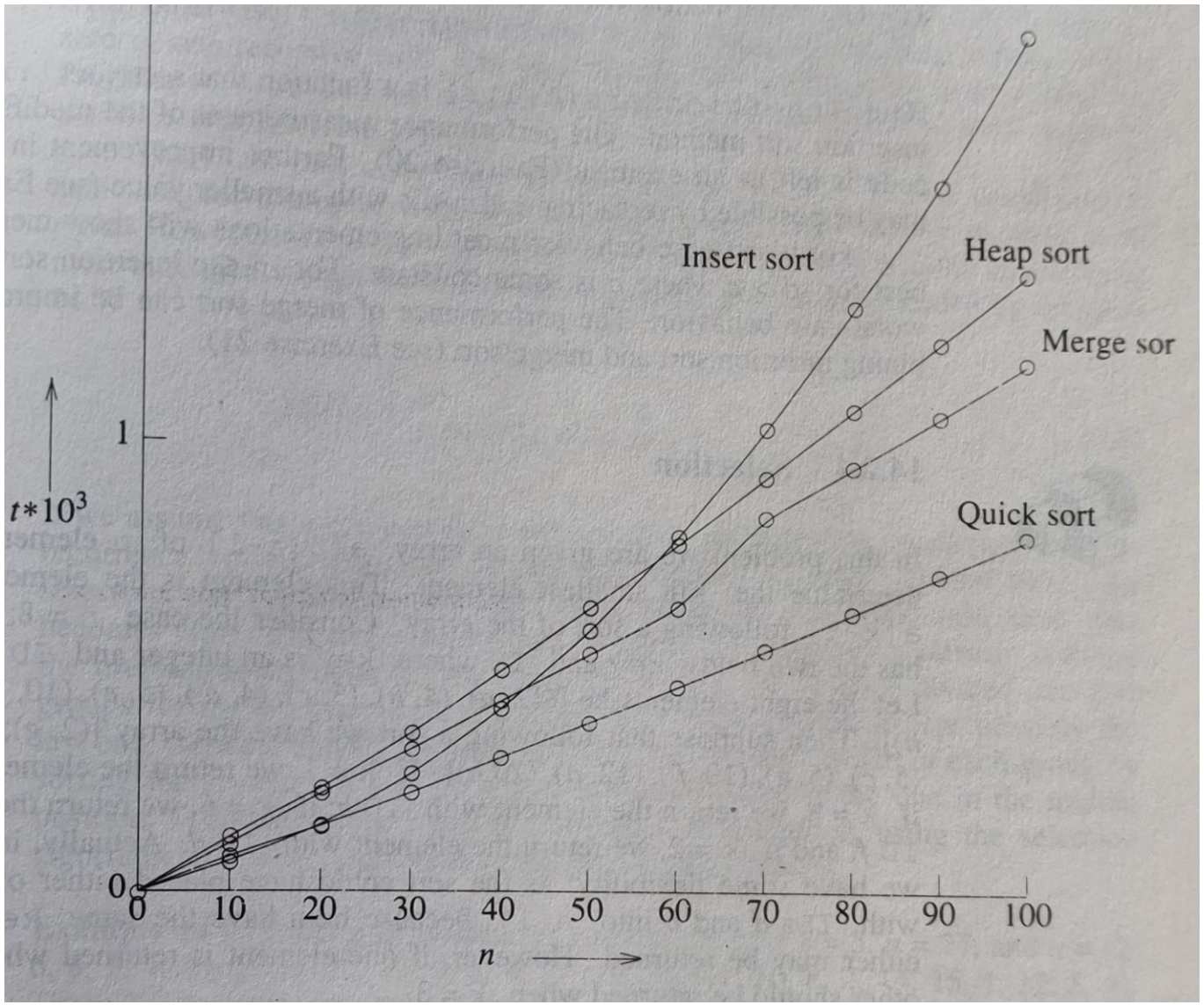
recursion tree →



Sorting Algorithm Comparison

Shell Sort → partitioning + insertion sort
↳ O(n log n)

- Selection sort algorithm is too simple, but performs poor and no optimization possible. ↳ no caching usability
- Bubble sort can be improved to reduce number of iterations.
- Insertion sort performs well if number of elements are too less. Good if adding elements and resorting.
- Quick sort is stable if number of elements increase. However worst case performance is poor.
- Merge sort also perform good, but need extra auxiliary space. O(n)



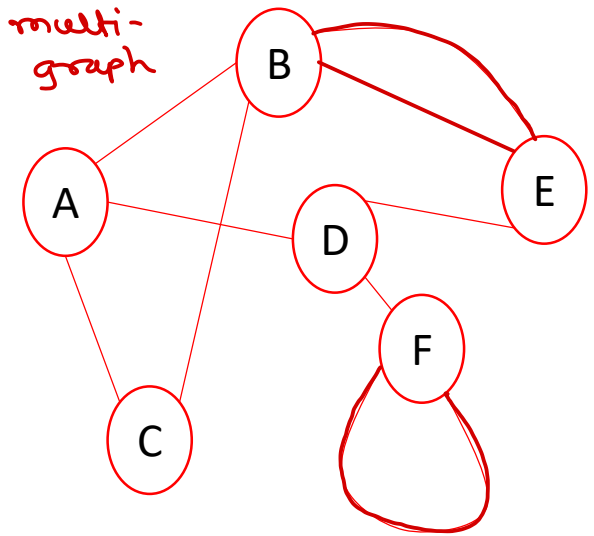
Graph Data Structure & Algorithms

Sunbeam Infotech

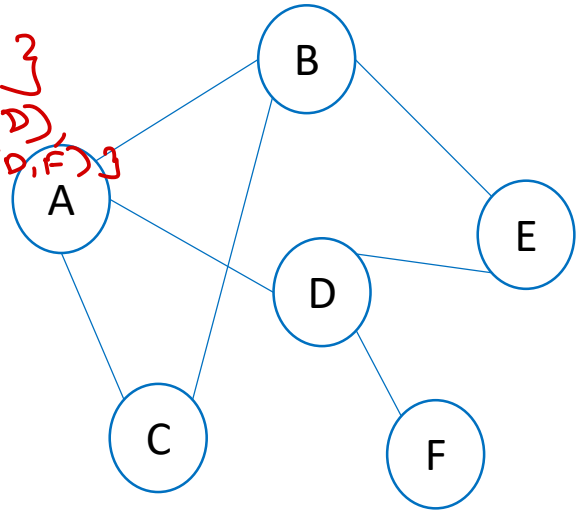
Graph

Graph Theory - Math

- Graph is a non-linear data structure.
- Graph is defined as set of vertices and edges. Vertices (also called as nodes) hold data, while edges connect vertices and represent relations between them.
  - G = { V, E }
- Vertices hold the data and Edges represents relation between vertices.
- When there is an edge from vertex P to vertex Q, P is said to be adjacent to Q.
- Multi-graph
  - Contains multiple edges in adjacent vertices or loops (edge connecting a vertex to it-self).
- Simple graph
  - Doesn't contain multiple edges in adjacent vertices or loops.

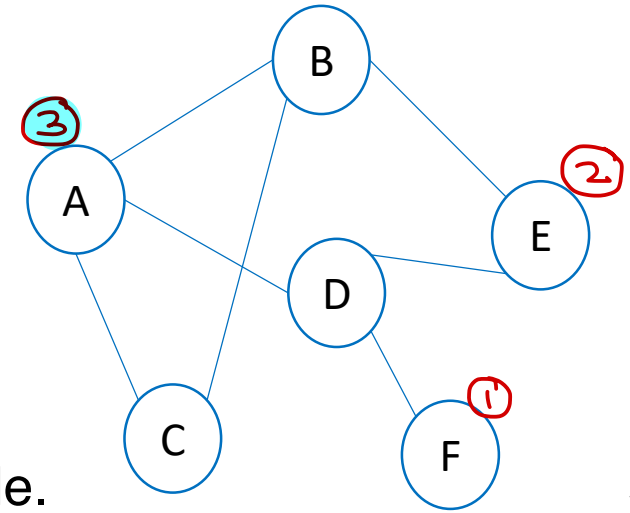


V = { A, B, C, D, E, F }
E = { (A,B), (A,C), (A,D), (B,C), (B,E), (D,E), (D,F) }

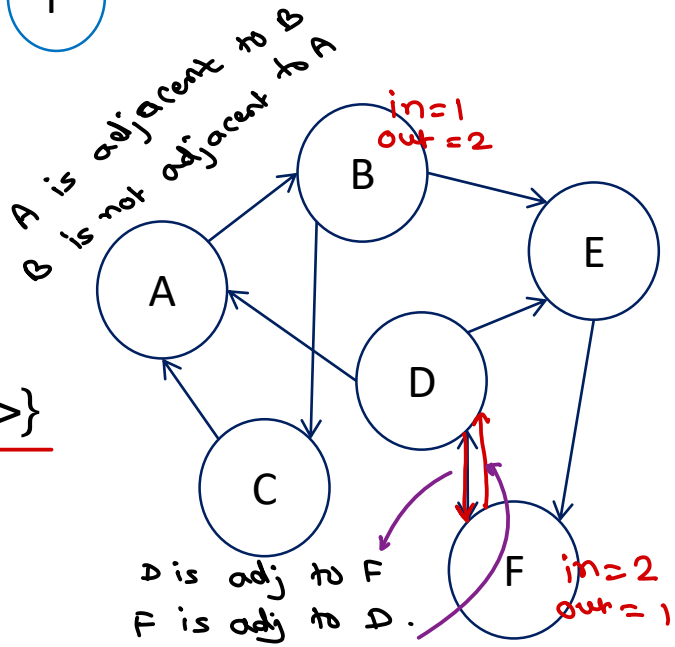


Graph

- Graph edges may or may not have directions.



- Undirected Graph: G = { V, E }
  - V = { A, B, C, D, E, F}
  - E = { (A,B), (A,C), (A,D), (B,C), (B,E), (D,E), (D,F) }
  - If P is adjacent to Q, then Q is also adjacent to P.
  - Degree of node: Number of nodes adjacent to the node.
  - Degree of graph: Maximum degree of any node in graph.

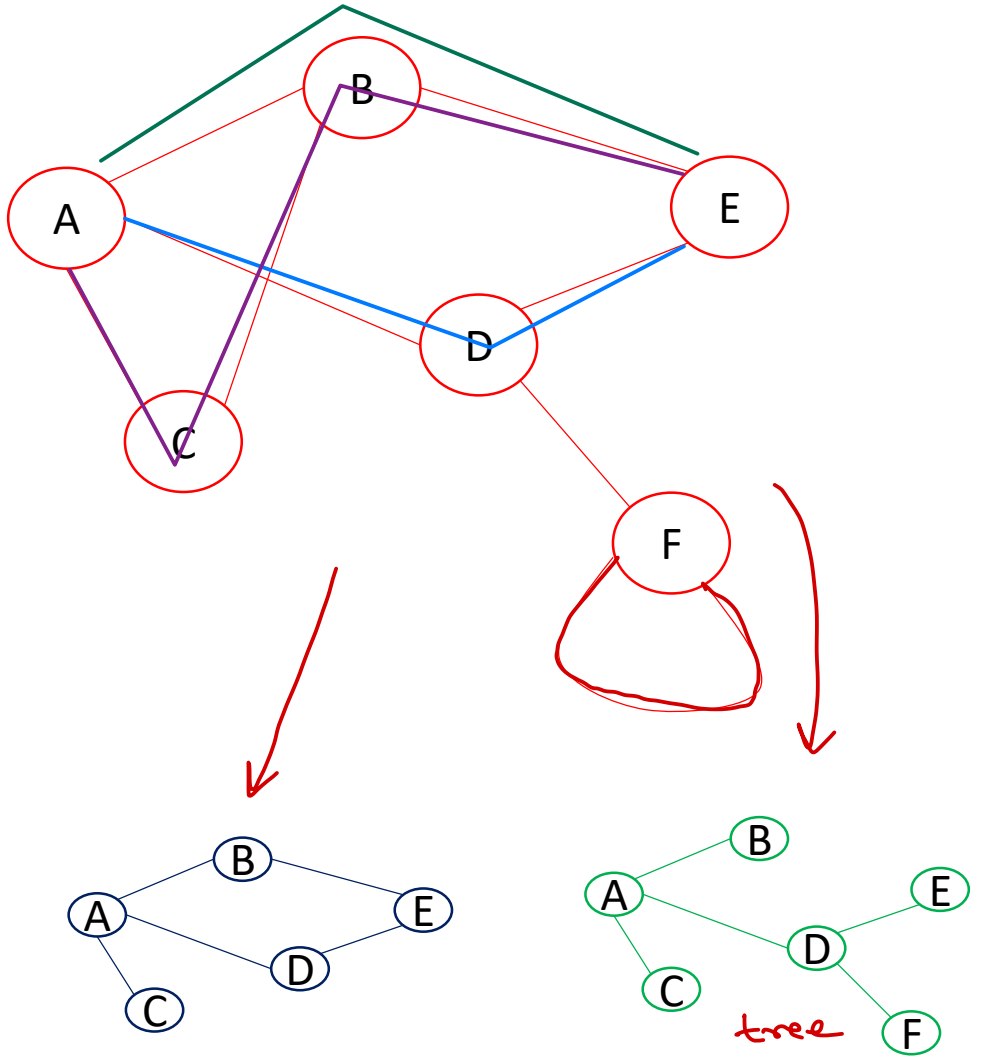


- Directed Graph: G = { V, E }
  - V = { A, B, C, D, E, F}
  - E = {<A,B>, <B,C>, <B,E>, <C,A>, <D,A>, <D,E>, <D,F>, <E,F>, <F,D>}
  - If P is adjacent to Q, then Q is may or may not be adjacent to P.
  - Out-degree: Number of edges originated from the node
  - In-degree: Number of edges terminated on the node

Graph

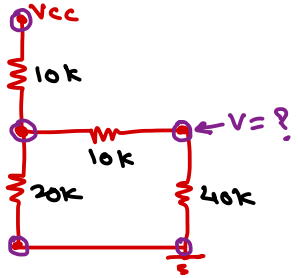
tree is graph without cycle.

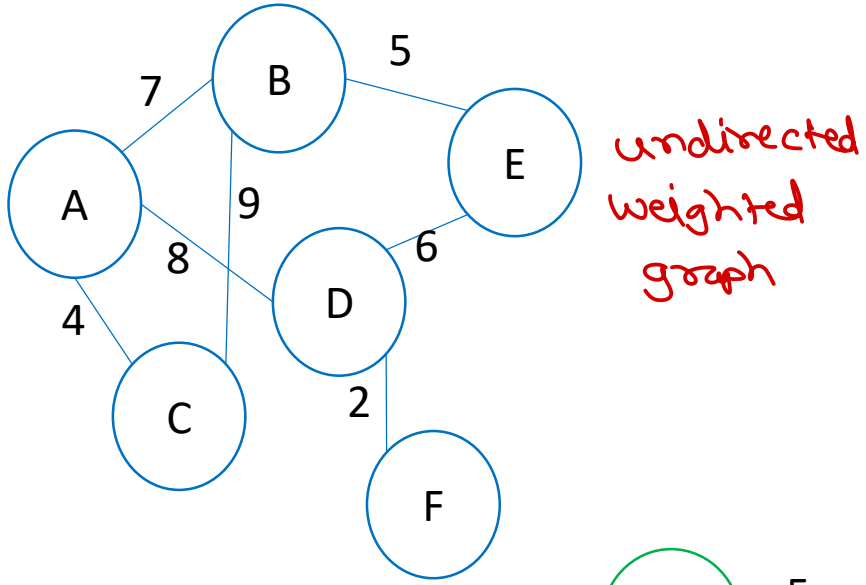
- Path: Set of edges between two vertices. There can be multiple paths between two vertices.
  - A – D – E
  - A – B – E
  - A – C – B – E
- Cycle: Path whose start and end vertex is same.
  - A – B – C – A
  - A – B – E – D – A
- Loop: Edge connecting vertex to itself. It is smallest cycle.
  - F – F
- Sub-Graph: A graph having few vertices and few edges in the given graph, is said to be sub-graph of given graph.

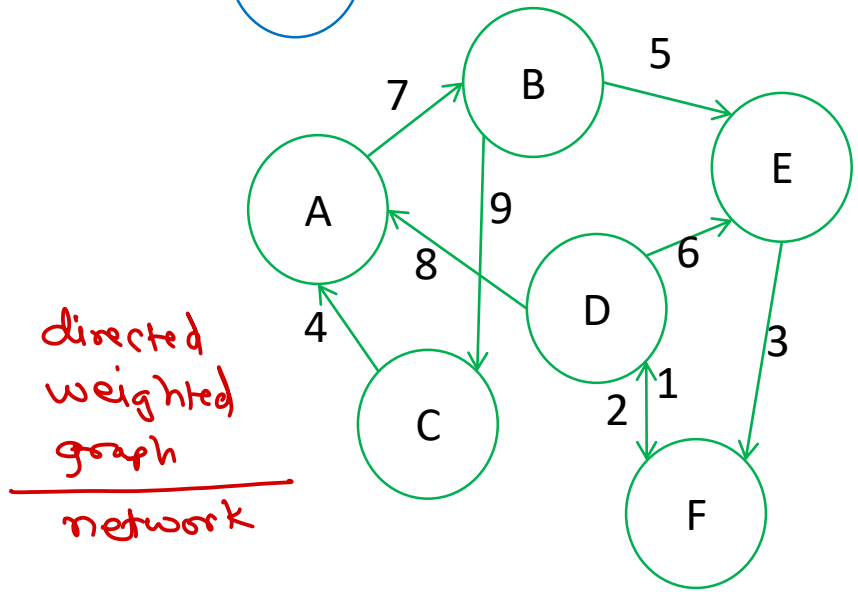


Graph

- Weighted graph
  - Graph edges have weight associated with them.
  - Weight represent some value e.g. distance, resistance.
- Directed Weighted graph (Network)
  - Graph edges have directions as well as weights.
- Applications of graph
  - Electronic circuits
  - Social media
  - Communication network
  - Road network
  - Flight/Train/Bus services
  - Bio-logical & Chemical experiments
  - Deep learning (Neural network, Tensor flow)
  - Graph databases (Neo4j)

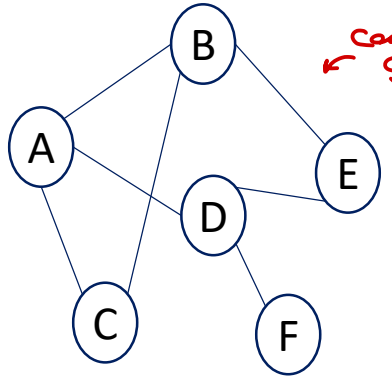




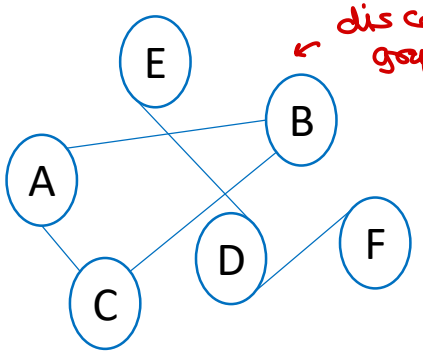


Graph

- Connected graph
  - From each vertex some path exists for every other vertex.
  - Can traverse the entire graph starting from any vertex.

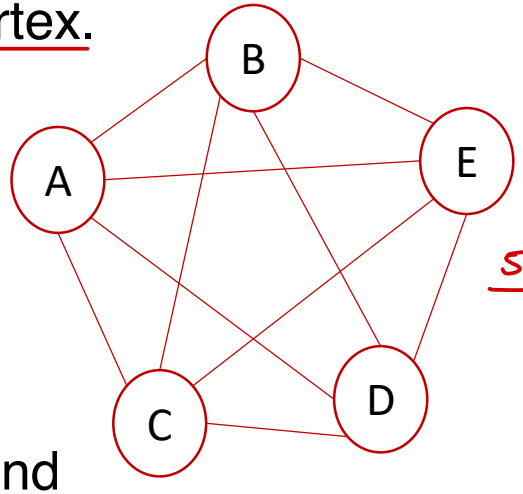


connected graph



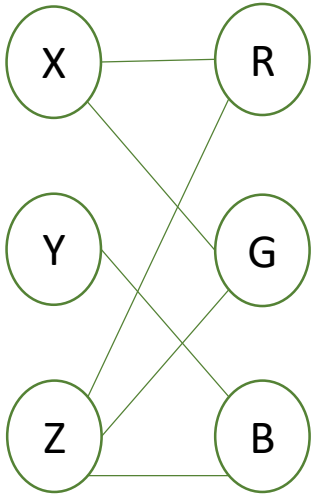
dis connected graph

- Complete graph
  - Each vertex of a graph is adjacent to every other vertex.
  - Un-directed graph: Number of edges = n (n-1) / 2
  - Directed graph: Number of edges = n (n-1)



$\frac{5 \times 4}{2} = 10$

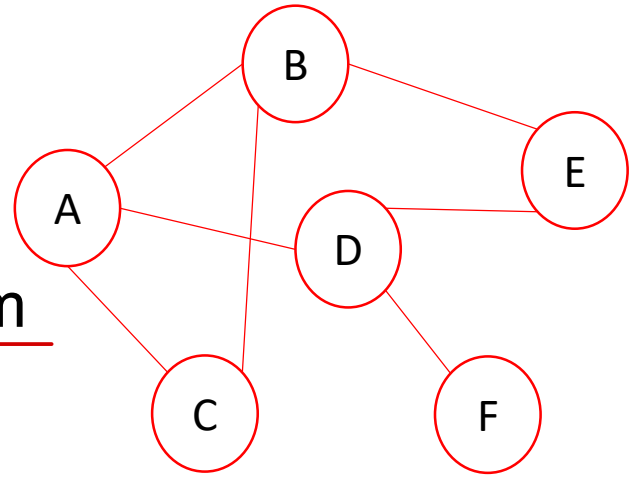
- Bi-partite graph
  - Vertices can be divided in two disjoint sets.
  - Vertices in first set are connected to vertices in second set.
  - Vertices in a set are not directly connected to each other.

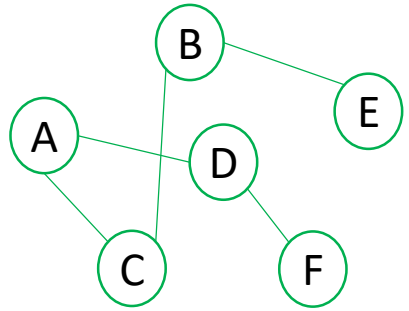


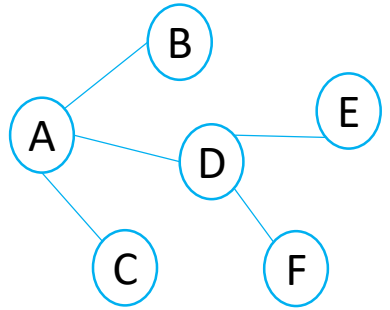
Spanning Tree

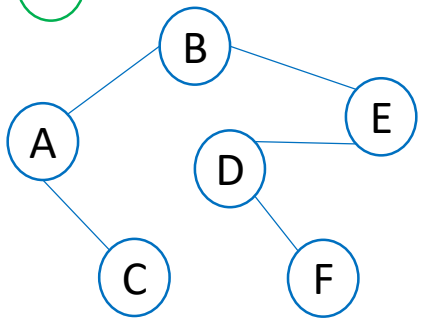
- Tree is a graph without cycles.
- Spanning tree is connected sub-graph of the given graph that contains all the vertices and sub-set of edges (V-1).
- Spanning tree can be created by removing few edges from the graph which are causing cycles to form.
- One graph can have multiple different spanning trees.
- In weighted graph, spanning tree can be made who has minimum weight (sum of weights of edges). Such spanning tree is called as Minimum Spanning Tree.
- Spanning tree can be made by various algorithms.
  - BFS Spanning tree
  - DFS Spanning tree
  - Prim's MST
  - Kruskal's MST

MST



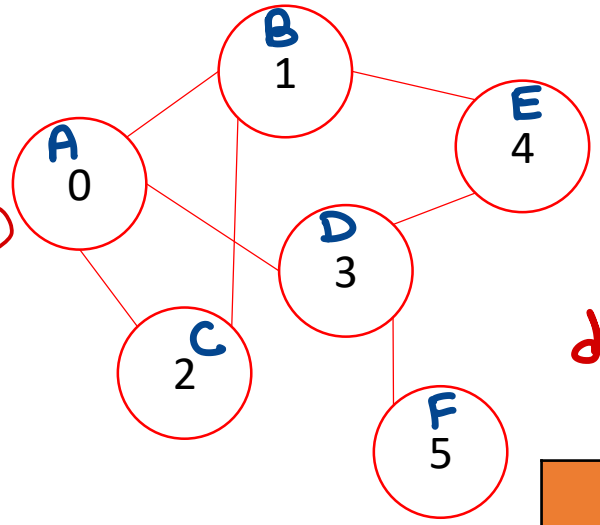






Graph Implementation – Adjacency Matrix

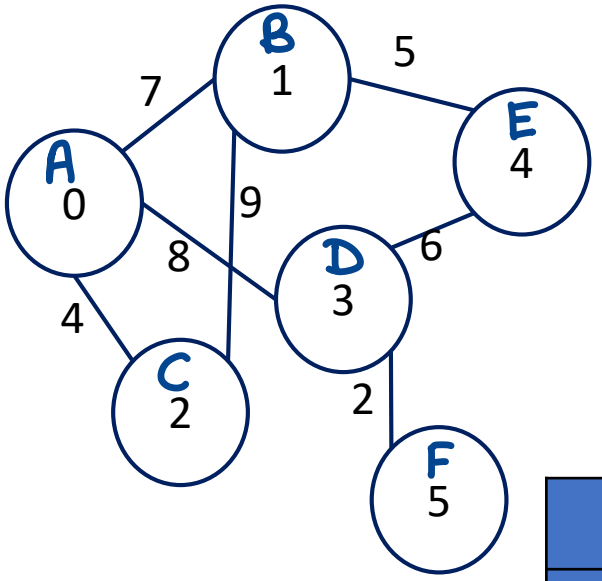
- If graph have V vertices, a V x V matrix can be formed to store edges of the graph.
- Each matrix element represent presence or absence of the edge between vertices.
- For *non-weighted graph*, 1 indicate edge and 0 indicate no edge.
- For un-directed graph, adjacency matrix is always symmetric across the diagonal.
- Space complexity of this implementation is $O(V^2)$.



	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	0	1	0
C	1	1	0	0	0	0
D	1	0	0	0	1	1
E	0	1	0	1	0	0
F	0	0	0	1	0	0

Graph Implementation – Adjacency Matrix

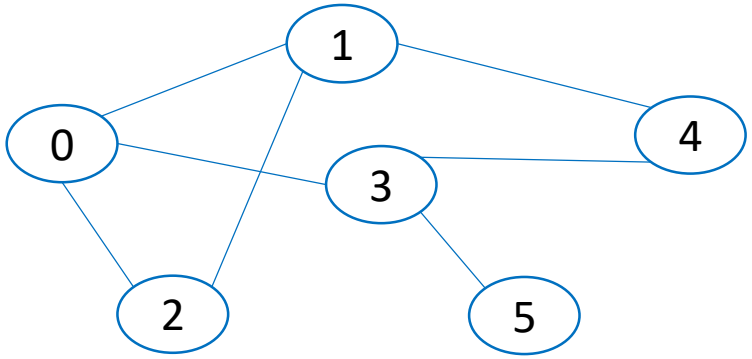
- If graph have V vertices, a V x V matrix can be formed to store edges of the graph.
- Each matrix element represent presence or absence of the edge between vertices.
- For *weighted graph*, weight value indicate the edge and infinity sign ∞ represent no edge.
- For un-directed graph, adjacency matrix is always symmetric across the diagonal.
- Space complexity of this implementation is $O(V^2)$.

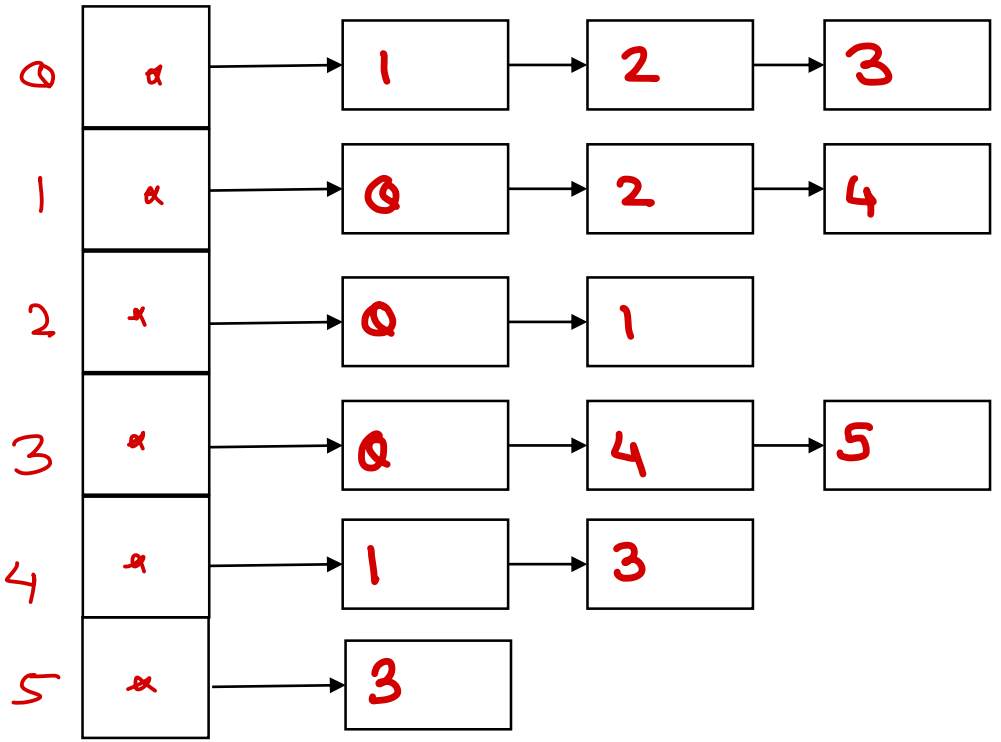


	A	B	C	D	E	F
A	∞	7	4	8	∞	∞
B	7	∞	9	∞	5	∞
C	4	9	∞	∞	∞	∞
D	8	∞	∞	∞	6	2
E	∞	5	∞	6	∞	∞
F	∞	∞	∞	2	∞	∞

Graph Implementation – Adjacency List

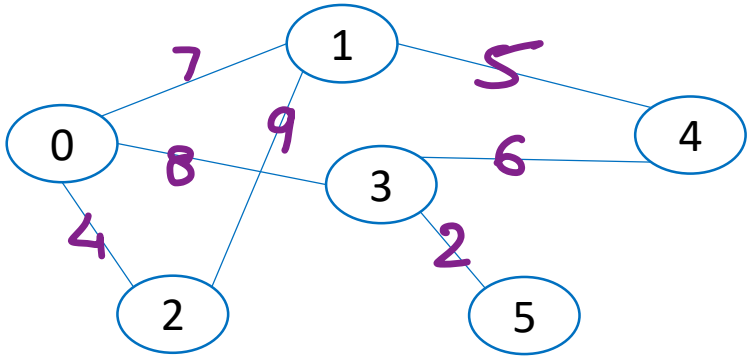
- Each vertex holds list of its adjacent vertices.
- For non-weighted graphs, only neighbour vertices are stored.
- For weighted graph, neighbour vertices and weights of connecting edges are stored.
- Space complexity of this implementation is O(V+E).
- If graph is sparse graph (with fewer number of edges), this implementation is more efficient (as compared to adjacency matrix method).

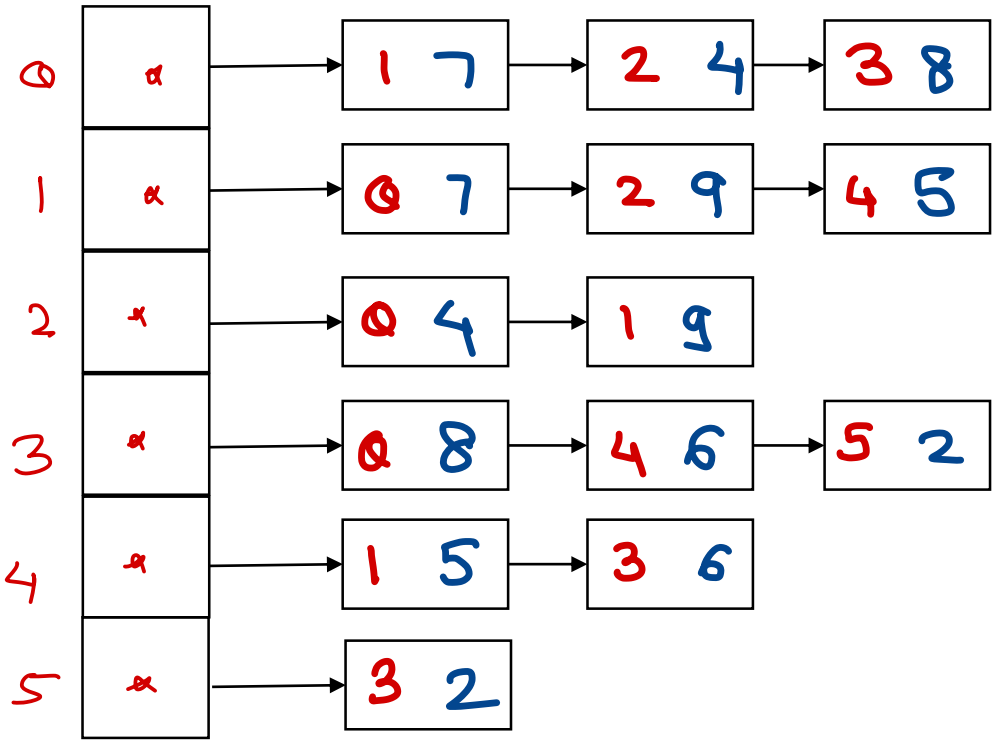




Graph Implementation – Adjacency List

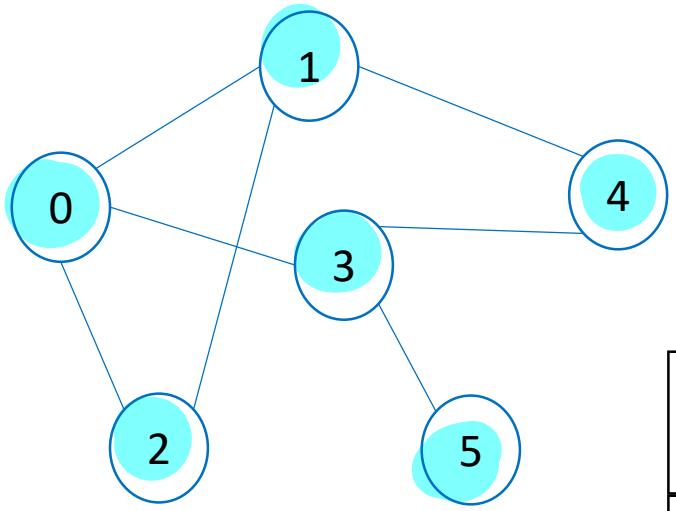
- Each vertex holds list of its adjacent vertices.
- For non-weighted graphs, only neighbour vertices are stored.
- For weighted graph, neighbour vertices and weights of connecting edges are stored.
- Space complexity of this implementation is O(V+E).
- If graph is sparse graph (with fewer number of edges), this implementation is more efficient (as compared to adjacency matrix method).



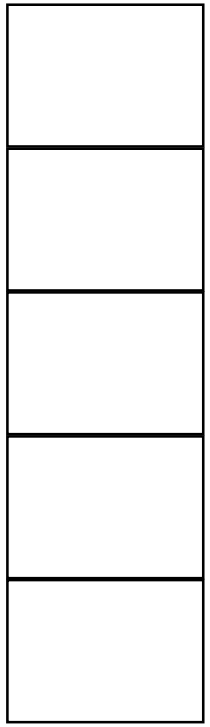


Graph Traversal – DFS Algorithm

1. Choose a vertex as start vertex.
2. Push start vertex on stack & mark it.
3. Pop vertex from stack.
4. Visit (Print) the vertex.
5. Put all non-~~visited~~ marked neighbours of the vertex on the stack and mark them.
6. Repeat 3-5 until stack is empty.



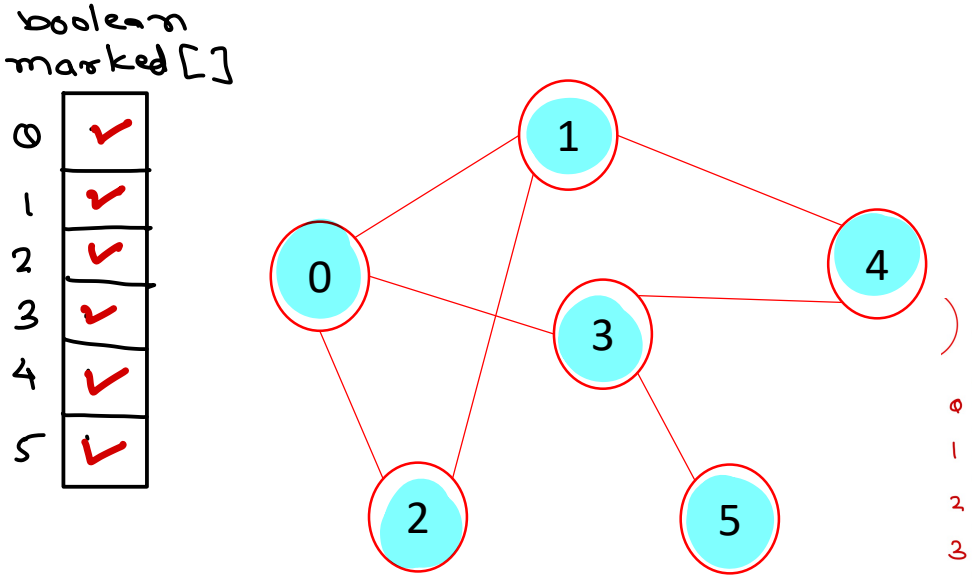
0 3 5 4 2 1



Graph Traversal – BFS Algorithm

1. Choose a vertex as start vertex.
2. Push start vertex on queue & mark it.
3. Pop vertex from queue.
4. Visit (Print) the vertex.
5. Put all non-~~visited~~ marked neighbours of the vertex on the queue and mark them.
6. Repeat 3-5 until queue is empty.

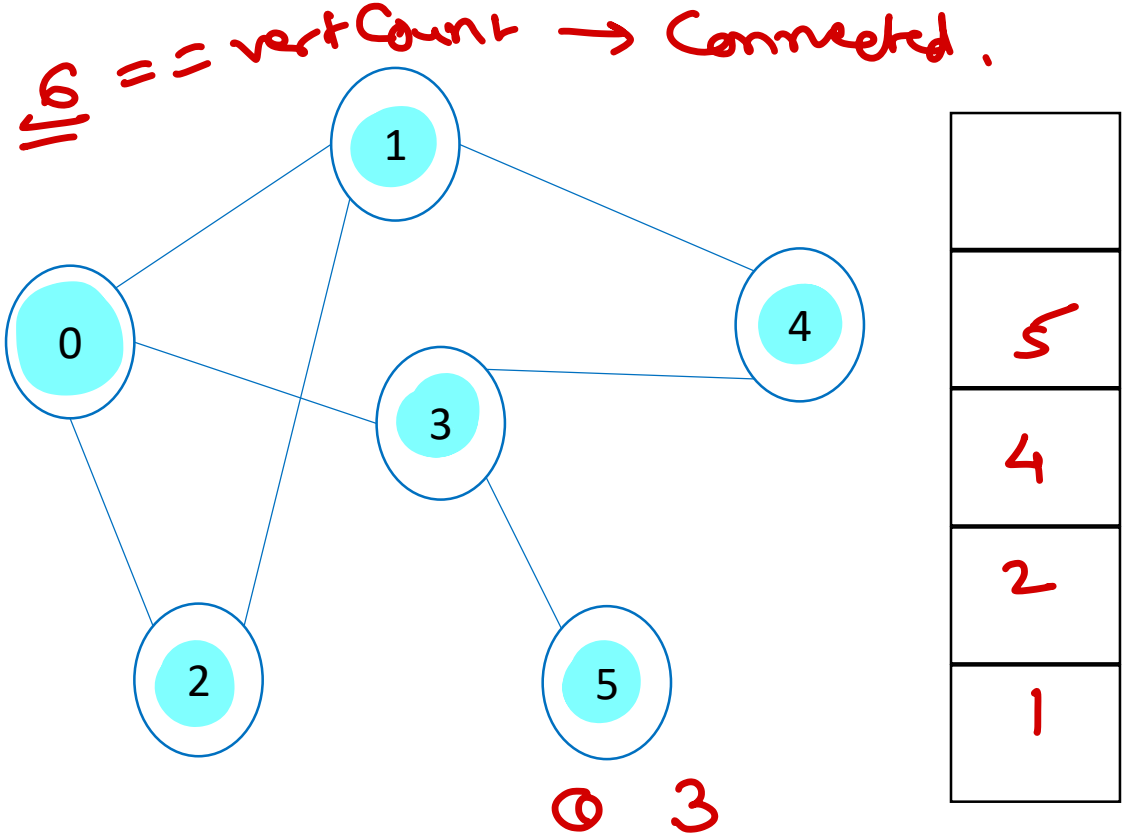
- BFS is also referred as level-wise search algorithm.



	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	0	1	0
C	1	1	0	0	0	0
D	1	0	0	0	1	1
E	0	1	0	1	0	0
F	0	0	0	1	0	0

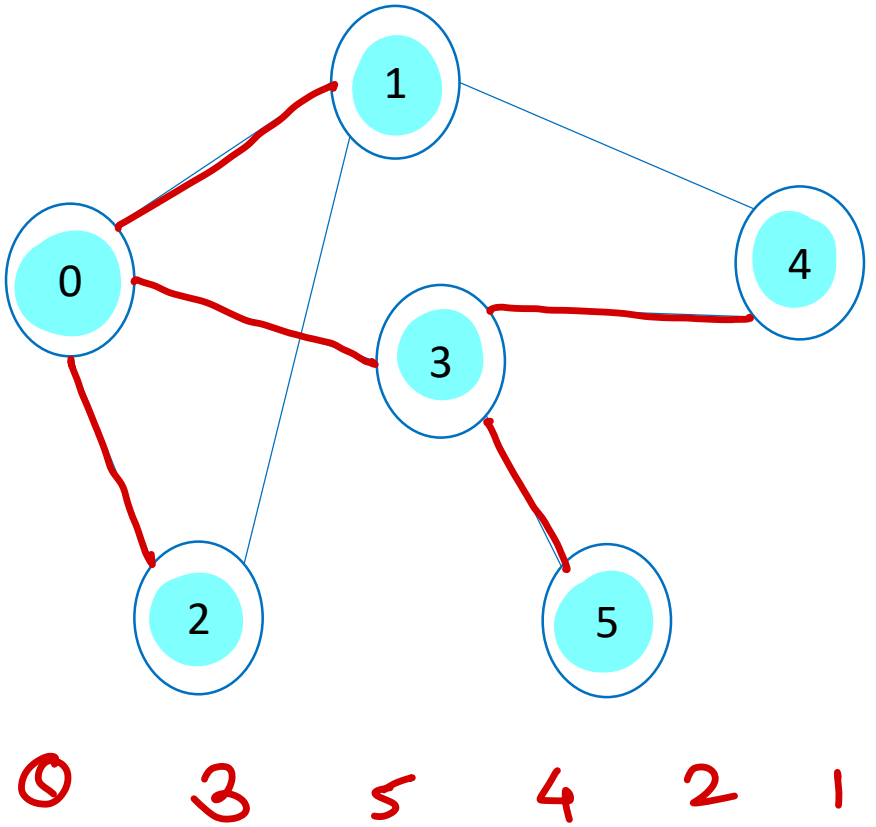
Check Connected-ness

1. push starting vertex on stack & mark it.
2. begin counting marked vertices from 1.
3. pop a vertex from stack.
4. push all its non-marked neighbors on the stack, mark them and increment count.
5. if count is same as number of vertices, graph is connected (return).
6. repeat steps 3-5 until stack is empty.
7. graph is not connected (return)



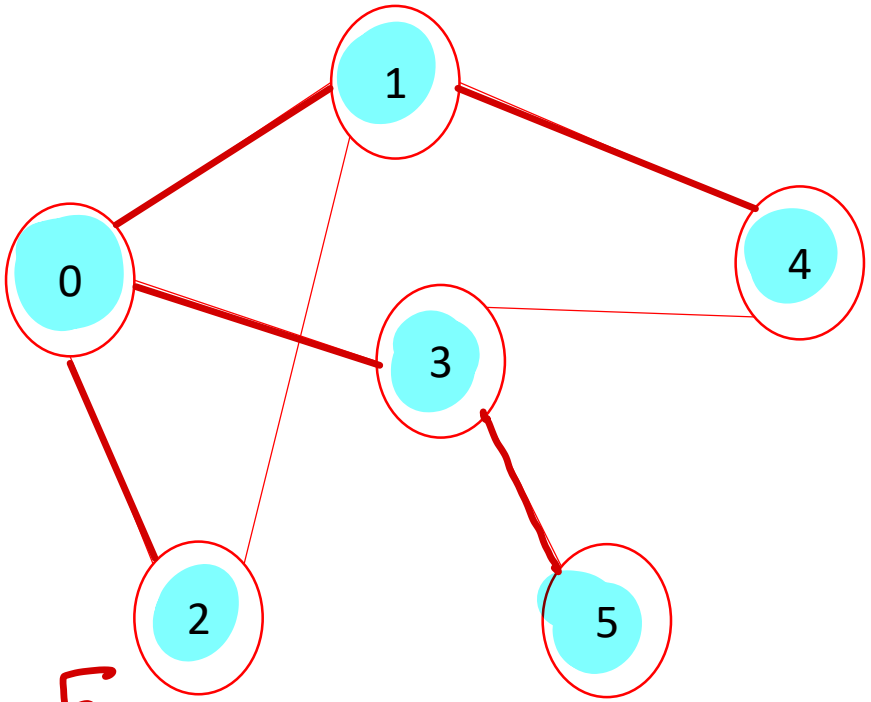
DFS Spanning Tree

1. push starting vertex on stack & mark it.
2. pop the vertex.
3. push all its non-marked neighbors on the stack, mark them. Also print the vertex to neighboring vertex edges.
4. repeat steps 2-3 until stack is empty.

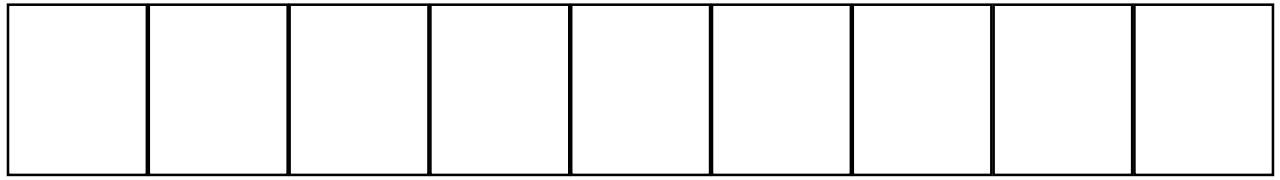


BFS Spanning Tree

1. push starting vertex on queue & mark it.
2. pop the vertex.
3. push all its non-marked neighbors on the queue, mark them. Also print the vertex to neighboring vertex edges.
4. repeat steps 2-3 until queue is empty.



0 1 2 3 4 5



Thank you!

Nilesh Ghule <Nilesh@sunbeaminfo.com>

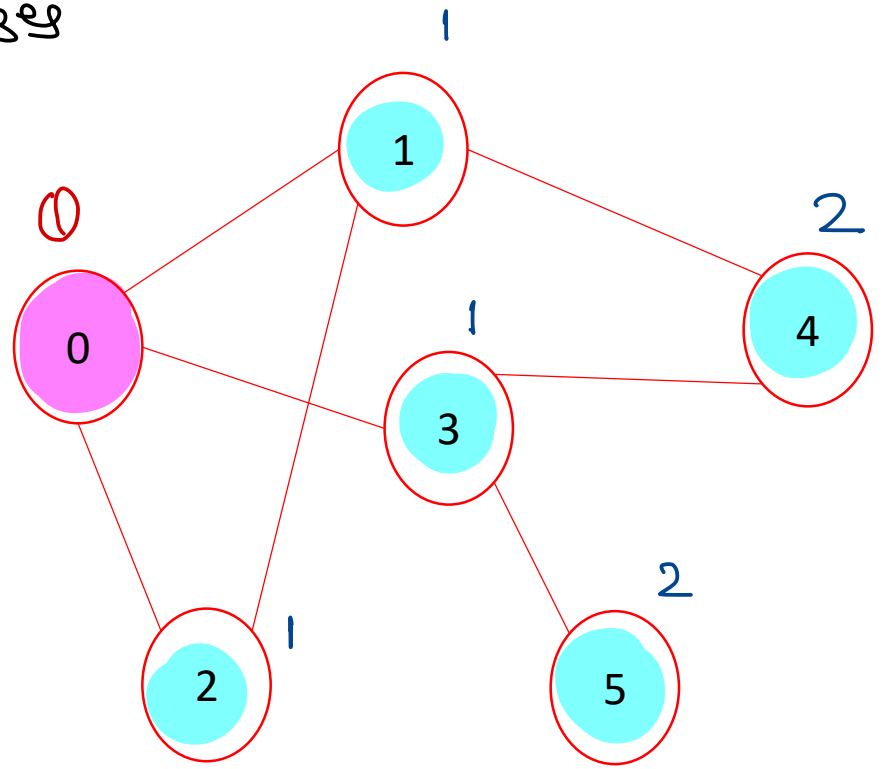
Graph Data Structure & Algorithms

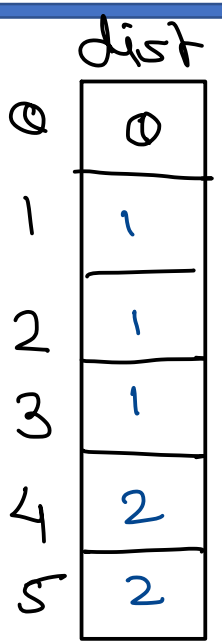
Sunbeam Infotech

Single Source Path Length - non-weighted graph

↳ from one vertex ↳ number of edges

1. Create path length array to keep distance of vertex from start vertex.
2. Consider dist of start vertex as 0.
3. push start vertex on queue & mark it.
4. pop the vertex.
5. push all its non-marked neighbors on the queue, mark them.
6. For each such vertex calculate its distance as dist[neighbor] = dist[current] + 1
7. repeat steps 3-6 until queue is empty.
8. Print path length array.

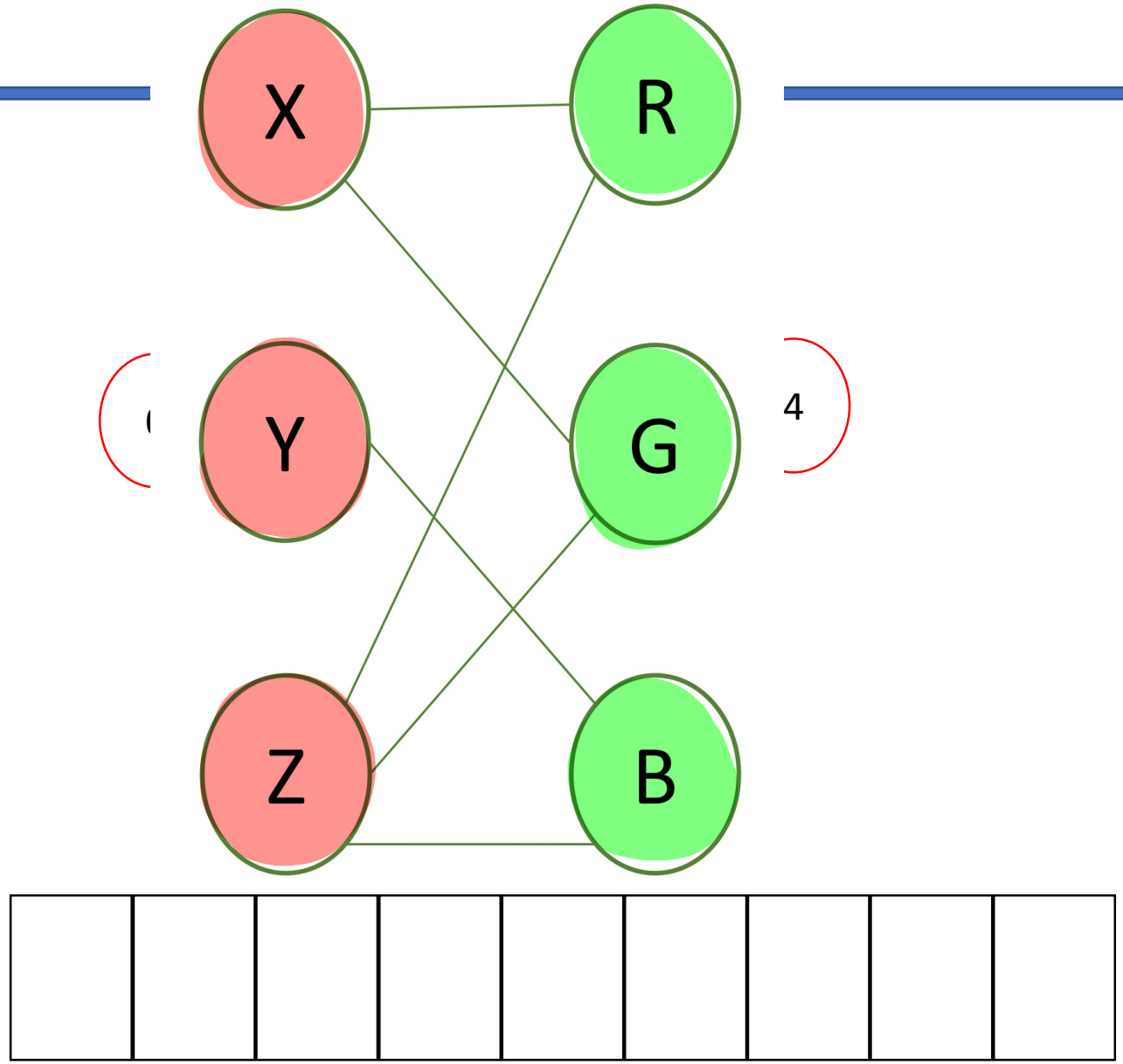




0 1 2 3 4 5

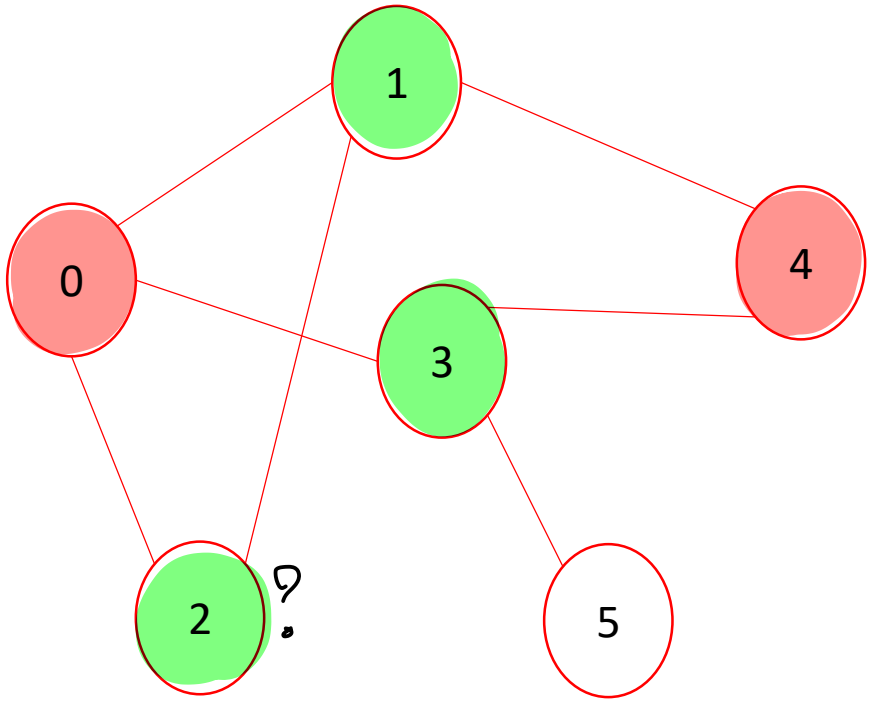
Check Bipartite-ness

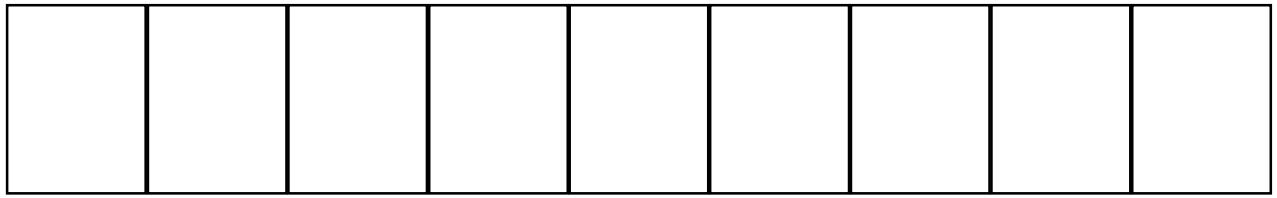
1. keep colors of all vertices in an array. Initially vertices have no color.
2. push start on queue & mark it. Assign it color1.
3. pop the vertex.
4. push all its non-marked neighbors on the queue, mark them.
5. For each such vertex if no color is assigned yet, assign opposite color of current vertex (c1-c2, c2-c1).
6. If vertex is already colored with same of current vertex, graph is not bipartite (return).
7. repeat steps 3-6 until queue is empty.



Check Bipartite-ness

1. keep colors of all vertices in an array. Initially vertices have no color.
2. push start on queue & mark it. Assign it color1.
3. pop the vertex.
4. push all its non-marked neighbors on the queue, mark them.
5. For each such vertex if no color is assigned yet, assign opposite color of current vertex (c1-c2, c2-c1).
6. If vertex is already colored with same of current vertex, graph is not bipartite (return).
7. repeat steps 3-6 until queue is empty.





Time Complexity - BFS / DFS

Adj Matrix Impl

outer loop → O(V)

inner loop → O(V)

Time Complexity → $O(V^2)$

Adj List Impl

outer loop → O(V)

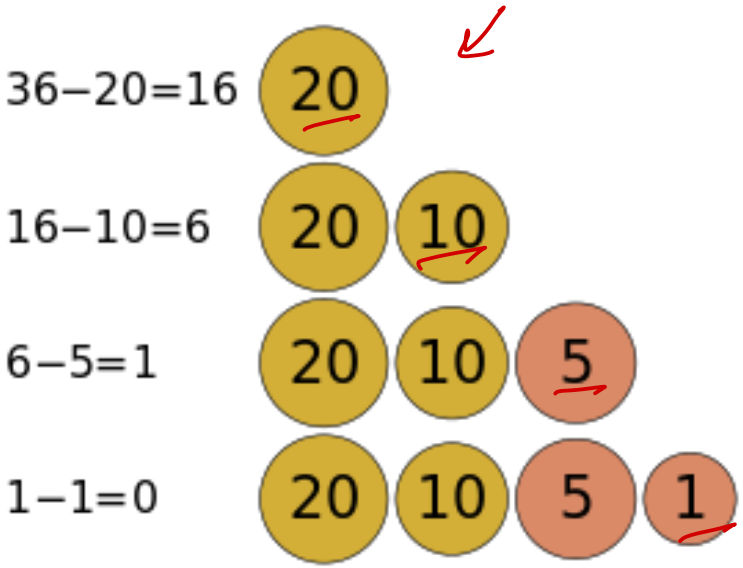
inner loop → for all vertices

O(E)

Time Complexity → O(V+E)

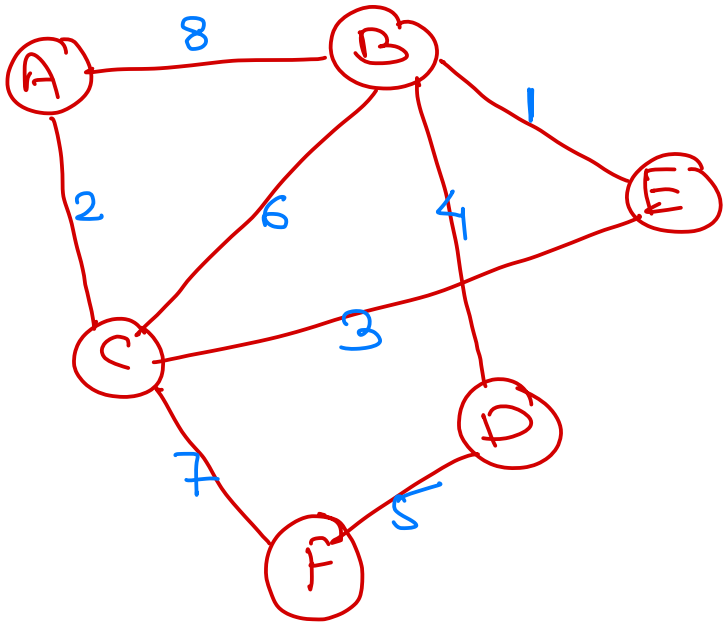
Problem solving technique: Greedy approach

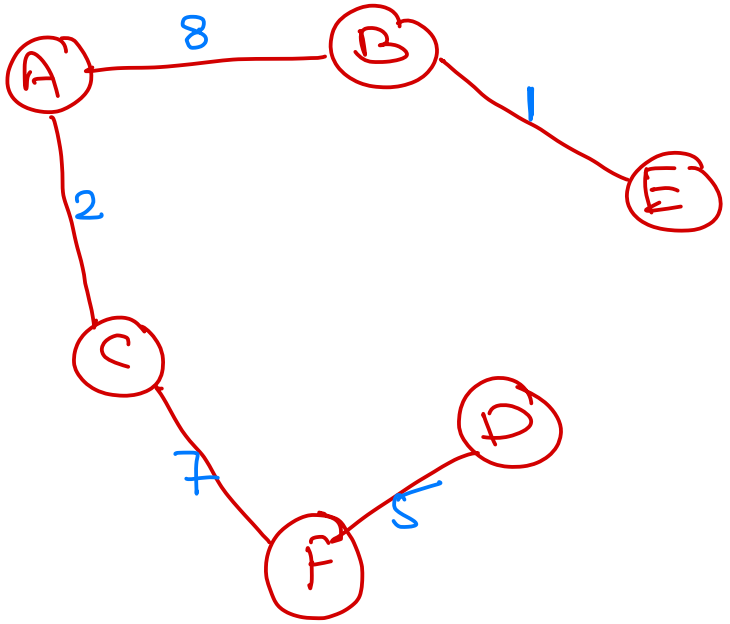
- A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.
- We can make choice that seems best at the moment and then solve the sub-problems that arise later.
- The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the sub-problem.
- It iteratively makes one greedy choice after another, reducing each given problem into a smaller one.
- A greedy algorithm never reconsiders its choices.
- A greedy strategy may not always produce an optimal solution.



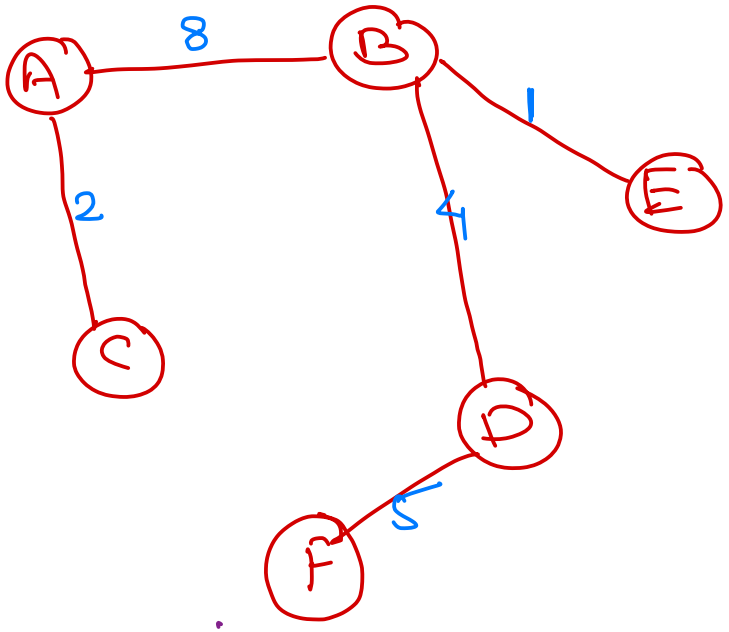
- Greedy algorithm decides minimum number of coins to give while making change.

Min Spanning Tree → spanning tree whose total weight is less than all other spanning trees.





weight = 23



weight = 20

Applications

* Optimal resource planning
* Travelling Salesman Problem
* Road making

Spanning Tree

V vertices

V-1 edges

no cycles

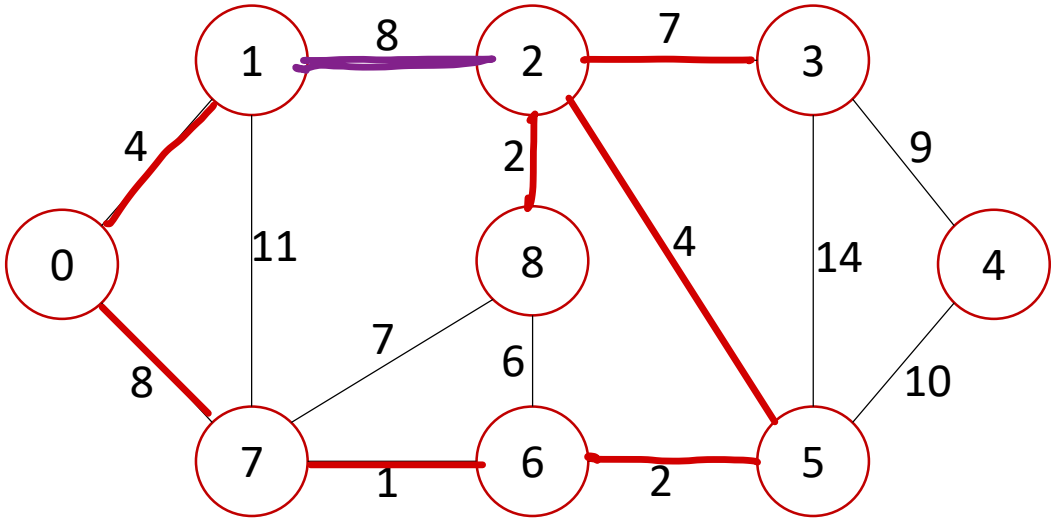
Algorithms

1. Kruskal
2. Prim

Union Find Algorithm

1. Consider all vertices as disjoint sets (parent = -1).
2. For each edge in the graph
   1. Find set of first vertex.
   2. Find set of second vertex.
   3. If both are in same set, cycle is detected.
   4. Otherwise, merge both the sets i.e. add root of first set under second set

```
parent[sr] = dr;
```



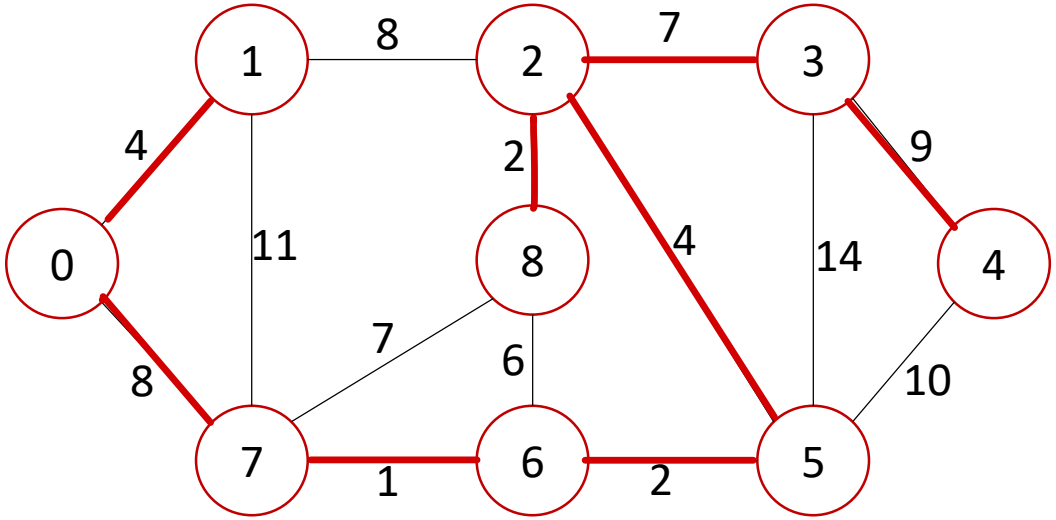
parent

1	3	5			3	5	6	2
0	1	2	3	4	5	6	7	8

src	des	wt
7	6	1
8	2	2
6	5	2
0	1	4
2	5	4
~~8~~	~~6~~	6
2	3	7
~~7~~	~~8~~	7
0	7	8
1	2	8
3	4	9
5	4	10
1	7	11
3	5	14

Kruskal's MST

1. Sort all the edges in ascending order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step 2 until there are (V-1) edges in the spanning tree.



src	des	wt
7	6	1
8	2	2
6	5	2
0	1	4
2	5	4
8	6	6
2	3	7
7	8	7
0	7	8
1	2	8
3	4	9
5	4	10
1	7	11
3	5	14

Thank you!

Nilesh Ghule <Nilesh@sunbeaminfo.com>

Graph Data Structure & Algorithms

Sunbeam Infotech

Kruskal's MST – Analysis

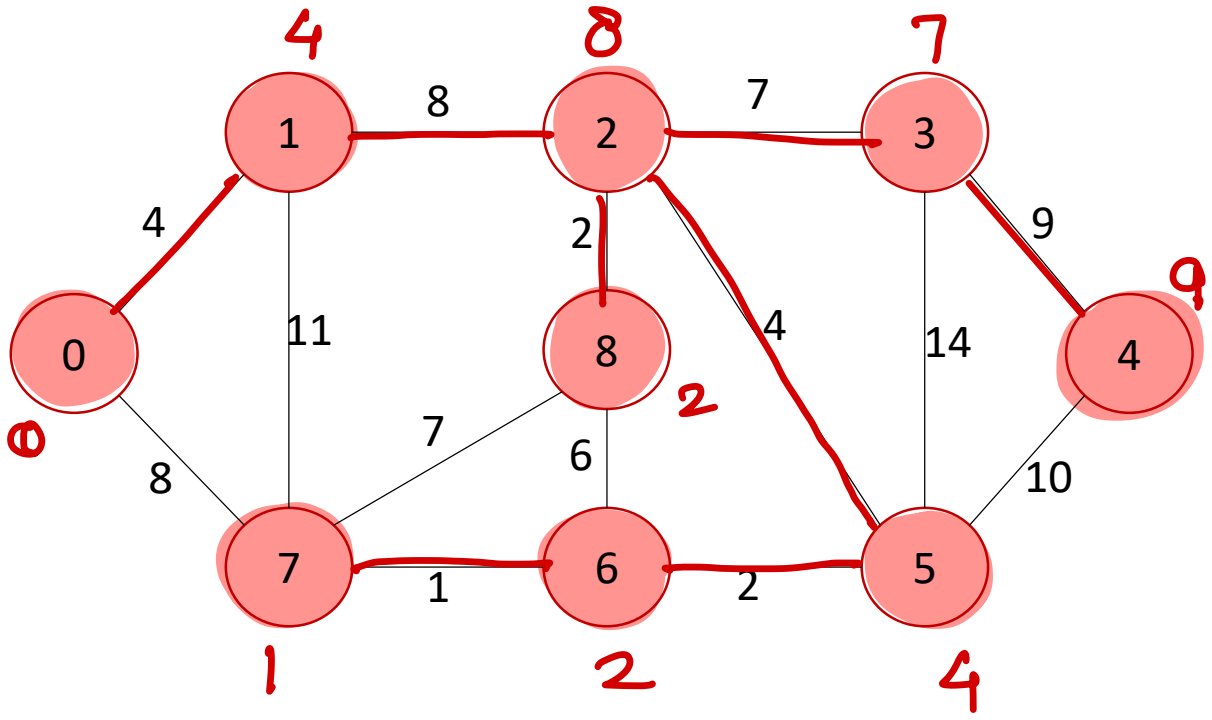
1. Sort all the edges in ascending order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step 2 until there are (V-1) edges in the spanning tree.

- Time complexity
  - Sort edges: O(E log E)
  - Pick edges (E edges): O(E)
  - Union Find: O(log V) – optimized.

- Time complexity
  - O(E log E + E log V)
  - E can max V^2.
  - So max time complexity: O(E log V).

Prim's MST

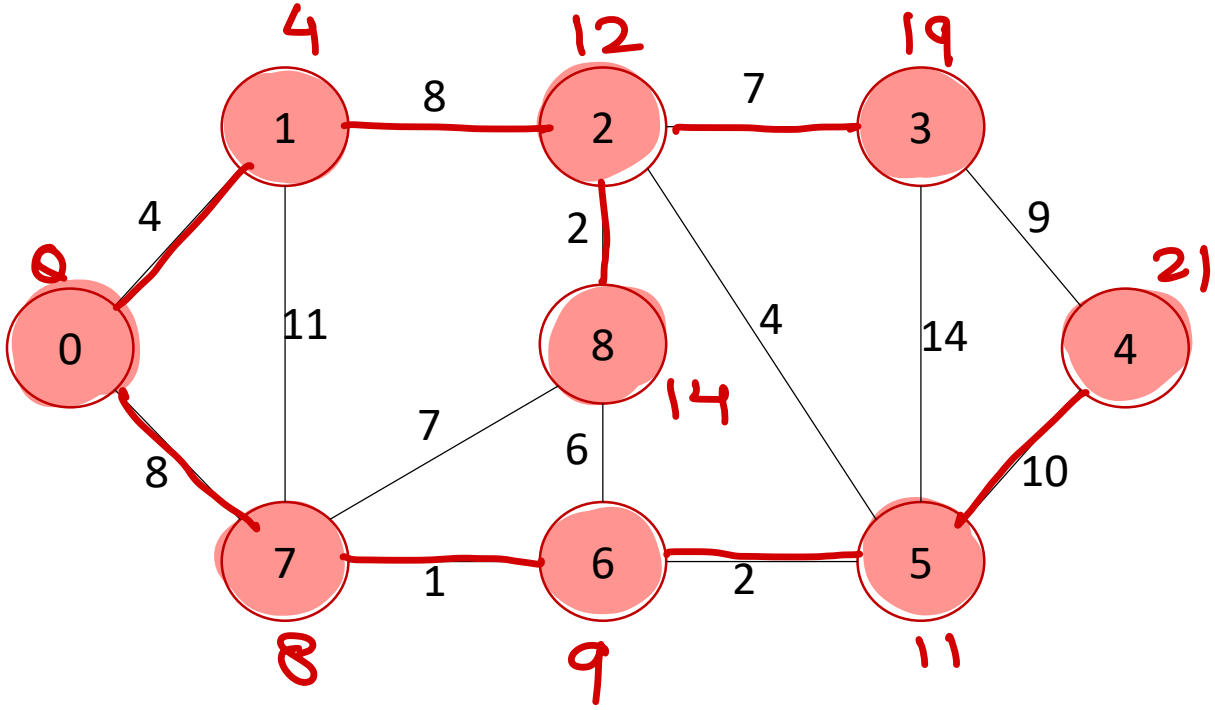
1. Create a set *mst* to keep track of vertices included in MST.
2. Also keep track of parent of each vertex. Initialize parent of each vertex -1.
3. Assign a key to all vertices in the input graph. Key for all vertices should be initialized to INF. The start vertex key should be 0.
4. While *mst* doesn't include all vertices
   i. Pick a vertex u which is not there in *mst* and has minimum key.
   ii. Include vertex u to *mst*.
   iii. Update key and parent of all adjacent vertices of u.
      a. For each adjacent vertex v, if weight of edge u-v is less than the current key of v, then update the key as weight of u-v.
      b. Record u as parent of v.



Dijkstra's Algorithm

→ Single Source Shortest Path also for weighted graphs.

1. Create a set *spt* to keep track of vertices included in shortest path tree.
2. Track distance of all vertices in the input graph. Distance for all vertices should be initialized to INF. The start vertex distance should be 0.
3. While *spt* doesn't include all vertices
   i. Pick a vertex u which is not there in *spt* and has minimum distance.
   ii. Include vertex u to *spt*.
   iii. Update distances of all adjacent vertices of u. For each adjacent vertex v, if distance of u + weight of edge u-v is less than the current distance of v, then update its distance as distance of u + weight of edge u-v.



Dijkstra's SPT – Analysis

1. Create a set *spt* to keep track of vertices included in shortest path tree.
2. Track distance of all vertices in the input graph. Distance for all vertices should be initialized to INF. The start vertex distance should be 0.
3. While *spt* doesn't include all vertices
   i. Pick a vertex u which is not there in *spt* and has minimum distance.
   ii. Include vertex u to *spt*.
   iii. Update distances of all adjacent vertices of u. For each adjacent vertex v, if distance of u + weight of edge u-v is less than the current distance of v, then update its distance as distance of u + weight of edge u-v.

- Time complexity (adjacency matrix)
  - V vertices: O(V)
  - get min key vertex: O(V)
  - update adjacent: O(V)
- Time complexity (adjacency matrix)
  - $O(V^2)$
- Time complexity (adjacency list)
  - V vertices: O(V)
  - get min key vertex: O(log V)
  - update adjacent: O(E) – E edges

E + V log V

- Time complexity (adjacency list)
  - O(E log V)

Dijkstra Algo.

dist[v] = dist[u] + weight(u,v)

① Cannot work with -ve edge weights.
e.g. chemical reaction (energy)
electronic ckt (current)

* solution → Bellman Ford. (SPT)

② Using dijkstra for finding shortest path to all vertices from each vertex :- all pair shortest path.

* Time complexity : O(V*(E+V log V))

* solution → Warshall Floyd.
(-ve weight)

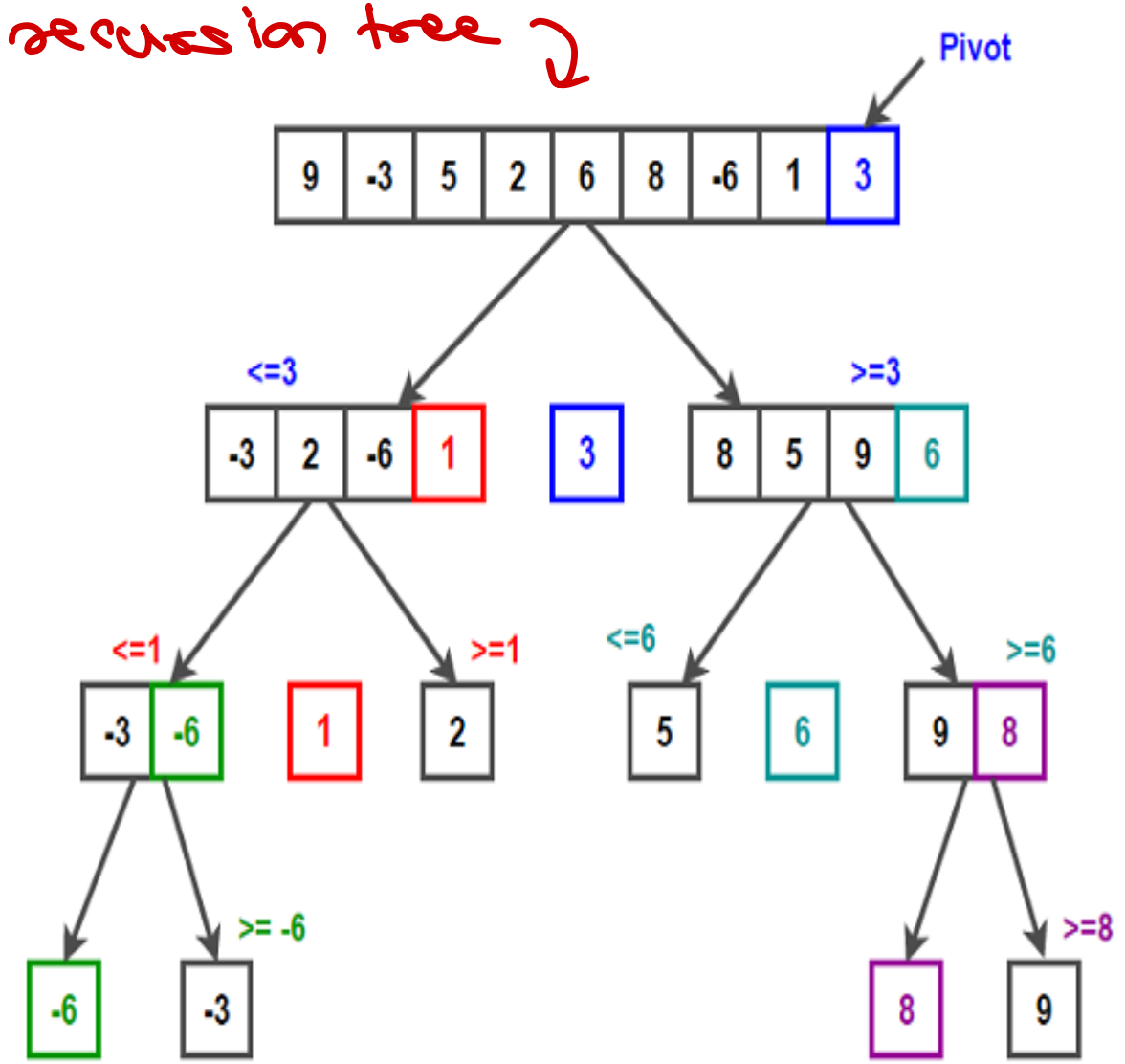
Dynamic Programming

Recursion

- Function calling itself is called as recursive function.
- For each function call stack frame is created on the stack.
- Thus it needs more space as well as more time for execution.
- However recursive functions are easy to program.
- Typical divide and conquer problems are solved using recursion.
- For recursive functions two things are must
  - Recursive call (Explain process it terms of itself)
  - Terminating or base condition (Where to stop)

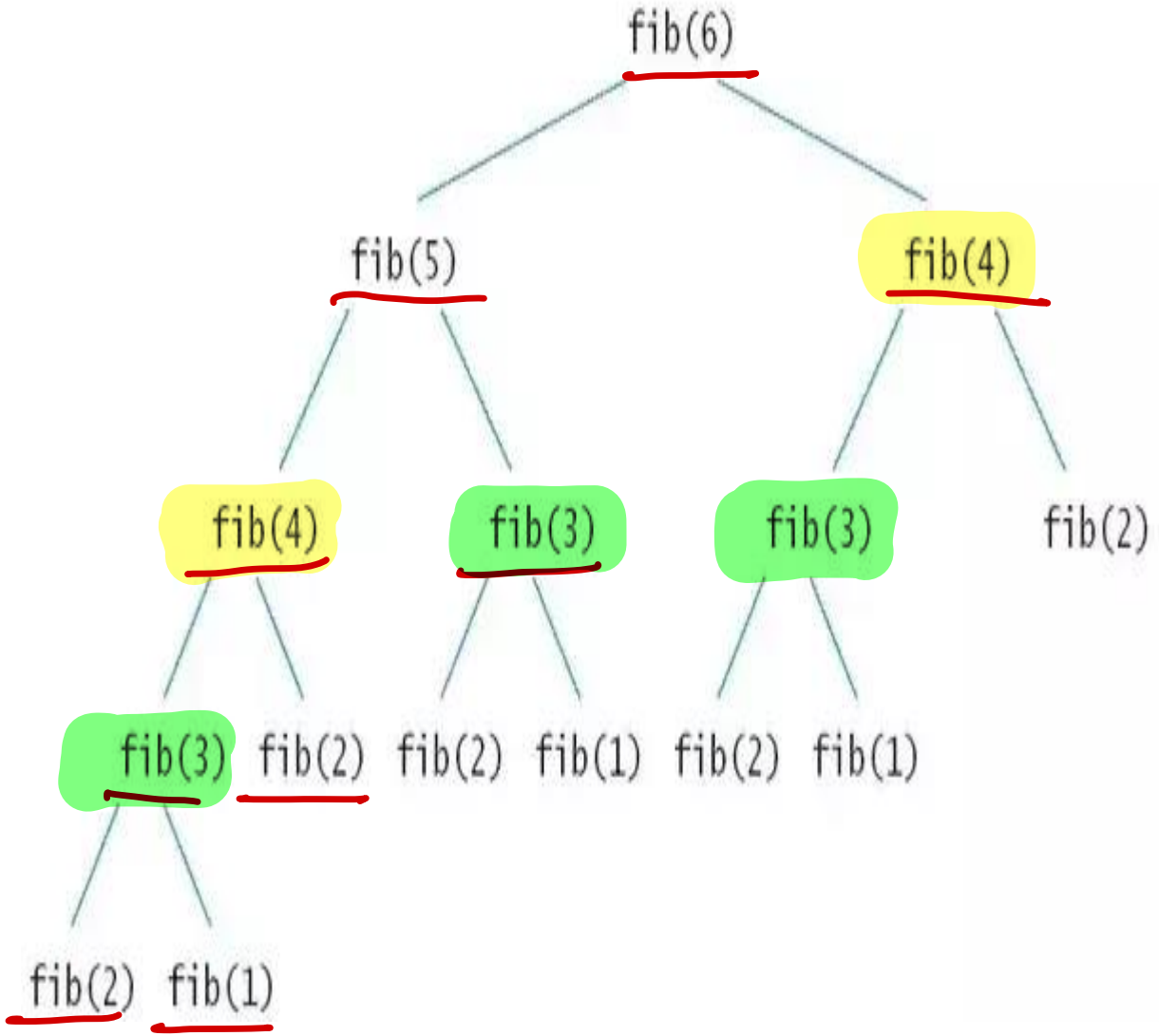
Recursion – QuickSort

- Algorithm
  1. If single element in partition, return.
  2. Last element as pivot.
  3. From left find element greater than pivot (x^{th} ele).
  4. From right find element less than pivot (y^{th} ele).
  5. Swap x^{th} ele with y^{th} ele.
  6. Repeat 2 to 4 until x < y.
  7. Swap y^{th} ele with pivot.
  8. Apply QuickSort to left partition (left to y-1).
  9. Apply QuickSort to right partition (y+1 to right).
- QS(arr, 0, 8)
  - QS(arr, 0, 3)
    - QS(arr, 0, 1)
      - QS(arr, 0, 0)
    - QS(arr, 3, 3)
  - QS(arr, 5, 8)
    - QS(arr, 5, 5)
    - QS(arr, 7, 8)
      - QS(arr, 9, 9)



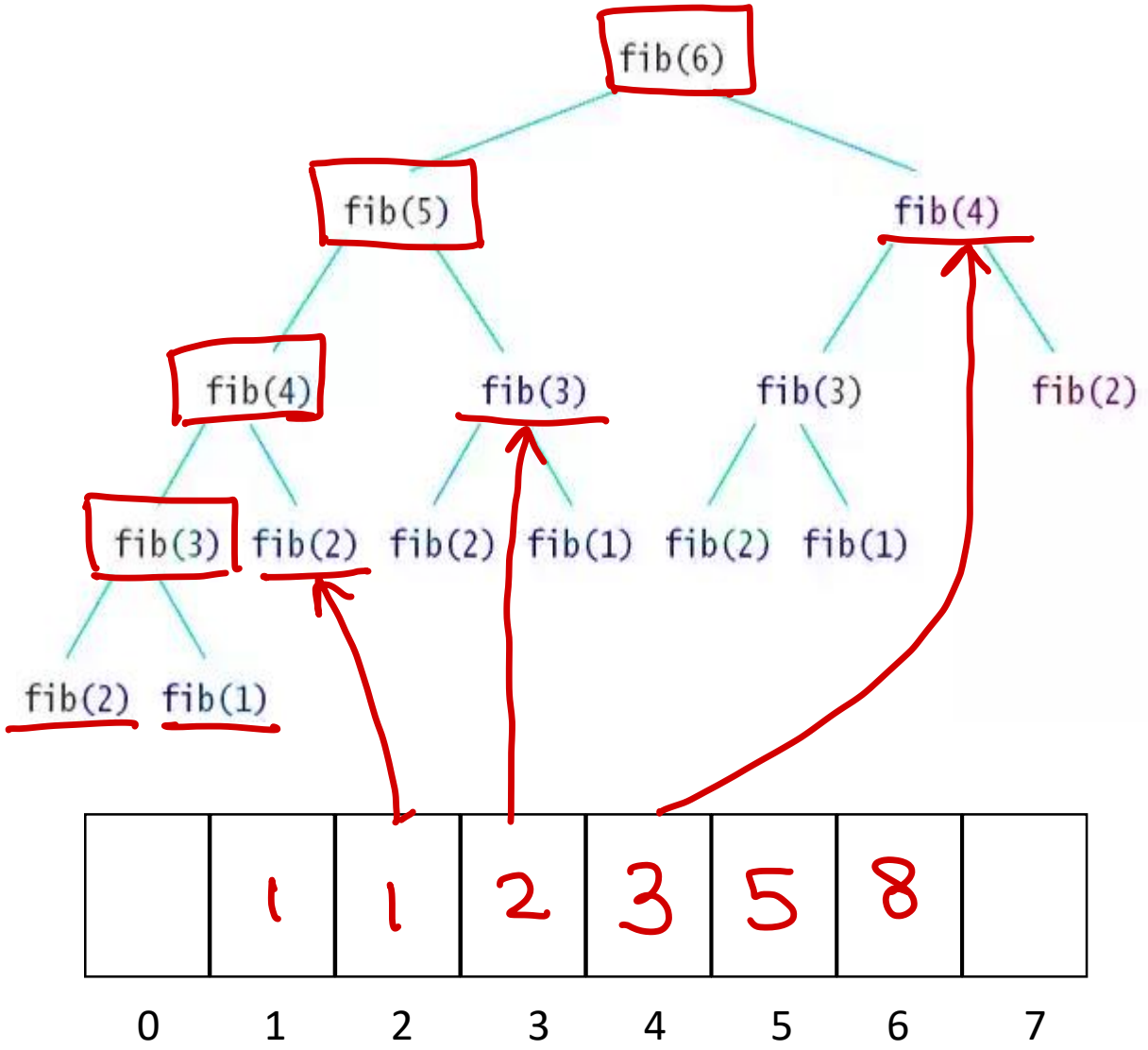
Recursion – Fibonacci Series

- Recursive formula
  - $T_n = T_{n-1} + T_{n-2}$
- Terminating condition
  - $T_1 = T_2 = 1$
- Overlapping sub-problem



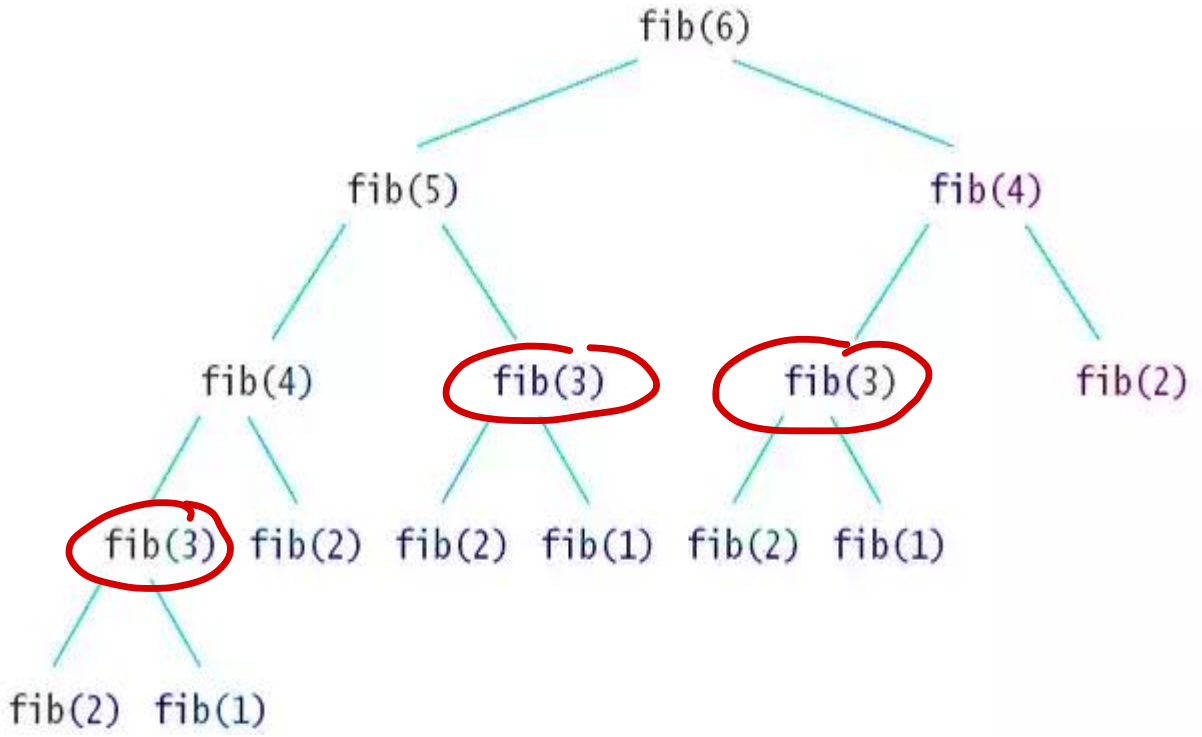
Memoization – Fibonacci Series

- It's based on the Latin word memorandum, meaning "to be remembered".
- Memoization is a technique used in computing to speed up programs.
- This is accomplished by memorizing the calculation results of processed input such as the results of function calls.
- If the same input or a function call with the same parameters is used, the previously stored results can be used again and unnecessary calculation are avoided.
- Need to rewrite recursive algorithm. Using simple arrays or map/dictionary.



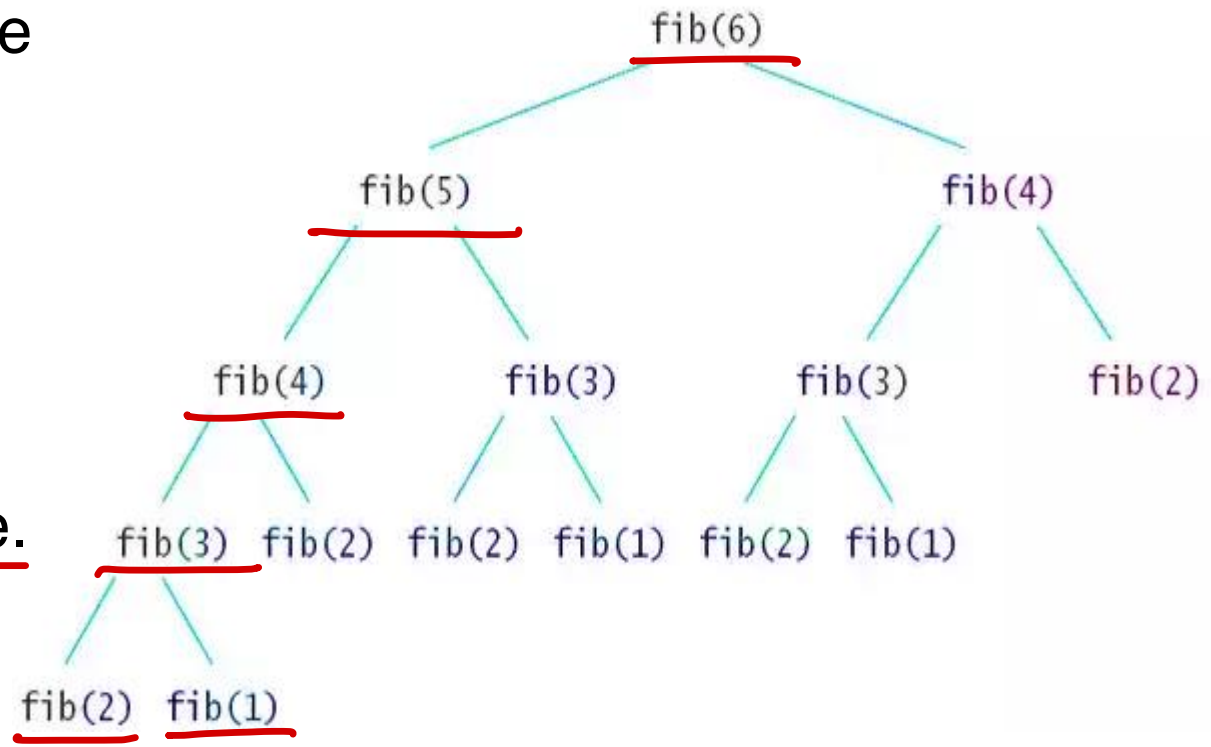
Dynamic Programming

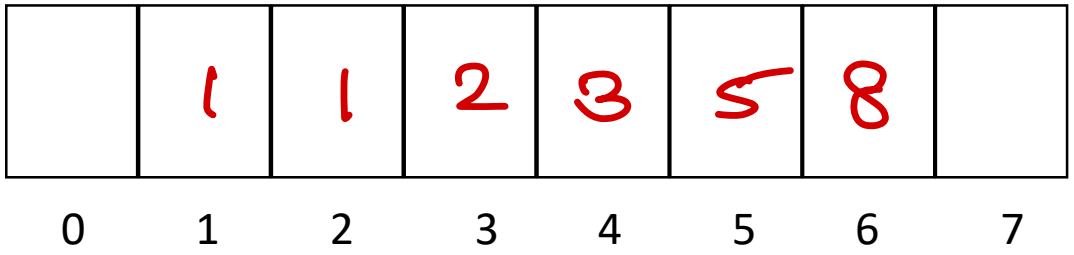
- Dynamic programming is another optimization over recursion.
- Typical DP problem give choices (to select from) and ask for optimal result (maximum or minimum).
- Technically it can be used for the problems having two properties
  - Overlapping sub-problems
  - Optimal sub-structure
- To solve problem, we need to solve its sub-problems multiple times.
- Optimal solution of problem can be obtained using optimal solutions of its sub-problems.



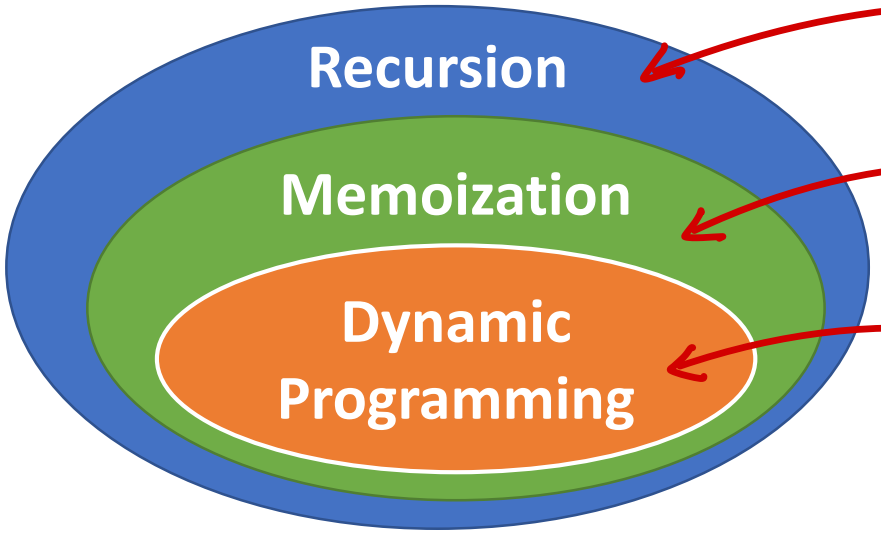
Dynamic Programming – Fibonacci Series

- Alternative solution to DP is memoizing the recursive calls. This solution needs more stack space, but similar in time complexity.
- Memoization is also referred as top-down approach.
- DP solution is bottom-up approach.
- DP use 1-d array or 2-d array to save state.
- Greedy algorithms pick optimal solution to local problem and never reconsider the choice done.
- DP algorithms solve the sub-problem in a iteration and improves upon it in subsequent iterations.





Dynamic Programming

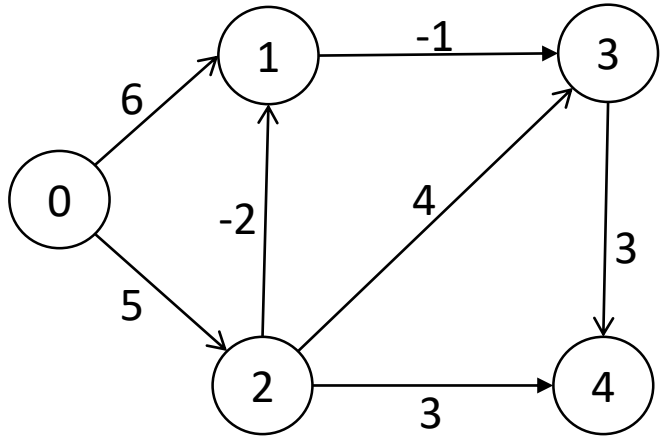


- Top down
- Optimized Top down – overlapping Sub-problems
- Bottom up – optimal substructure

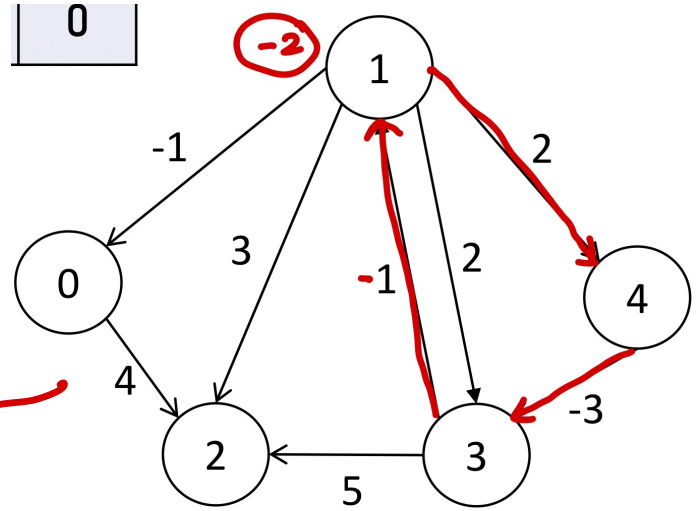
Bellman Ford Algorithm → Single source shortest path algo -ve weight edges.

- Initializes distances from the source to all vertices as infinite and distance to the source itself as 0.
- Calculates shortest distance V-1 times: For each edge u-v, if dist[v] > dist[u] + weight of edge u-v, then update dist[v], so that dist[v] = dist[u] + weight of edge u-v.
- Check if negative ~~edge~~ cycle in the graph: For each edge u-v, if dist[v] > dist[u] + weight of edge uv, then graph has –ve weight cycle.

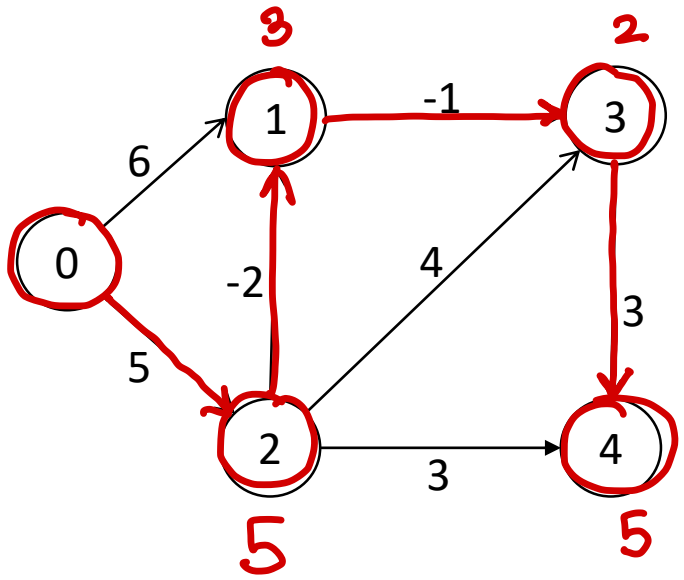
→ throw exception.

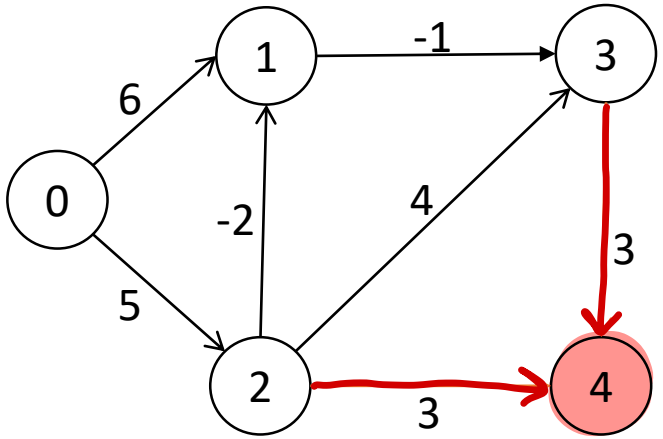


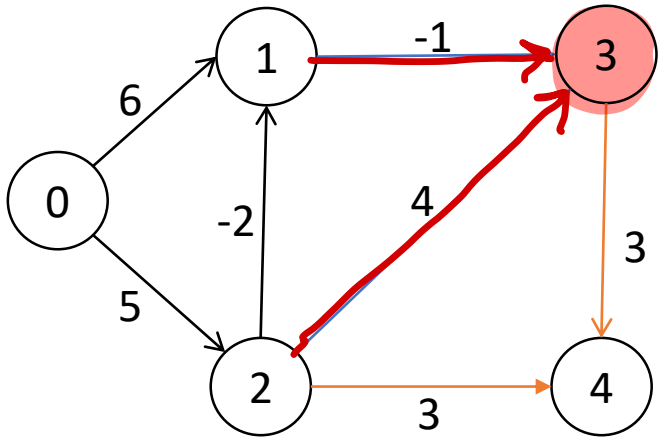
Src	Des	Wt
3	4	3
2	4	3
2	3	4
2	1	-2
1	3	-1
0	2	5
0	1	6

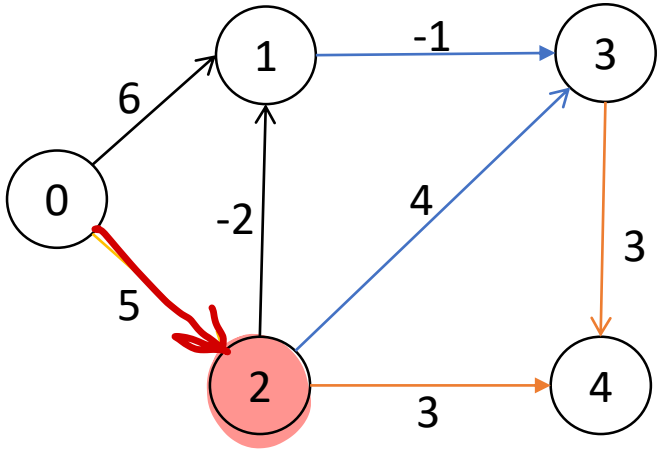


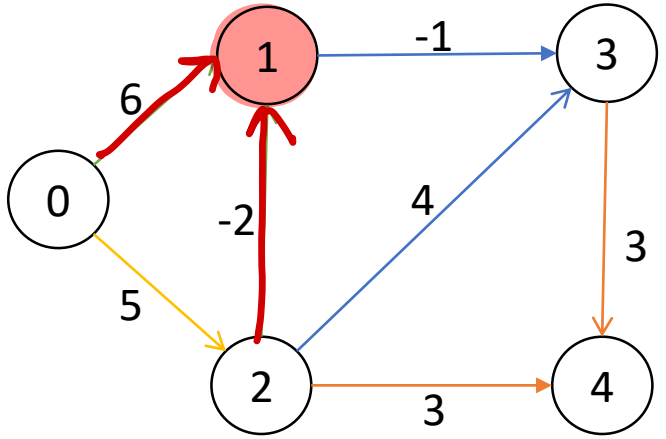
Bellman Ford Algorithm







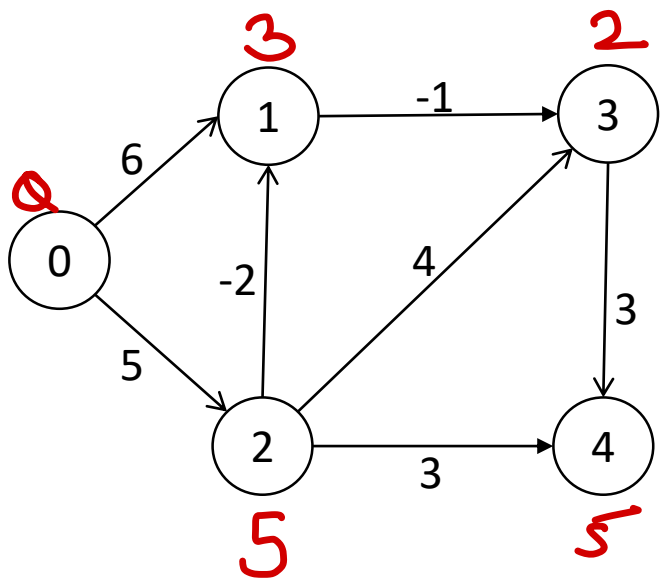




Bellman Ford Algorithm

O(VE)

	0	1	2	3	4
Pass 0	0	∞	∞	∞	∞
Pass 1	0	6	5	∞	∞
Pass 2	0	3	5	2	8
Pass 3	0	3	5	2	5
Pass 4	0	3	5	2	5



Src	Dest	Wt
3	4	3
2	4	3
2	3	4
2	1	-2
1	3	-1
0	2	5
0	1	6

Bellman Ford doesn't work with -ve weight cycle. In that case dist of vertices keep changing even after V-1 passes.

Warshall Floyd Algorithm - all pair shortest path

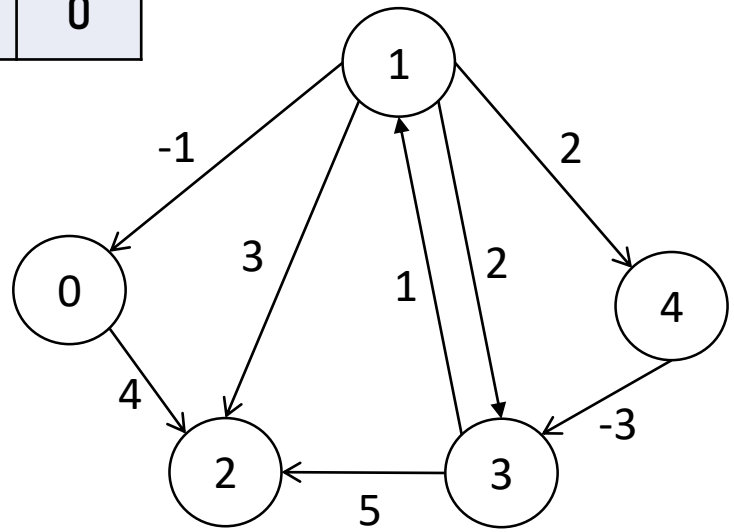
- Algorithm

1. Create distance matrix to keep distance of every vertex from each vertex. Initially assign it with weights of all edges among vertices (i.e. adjacency matrix).

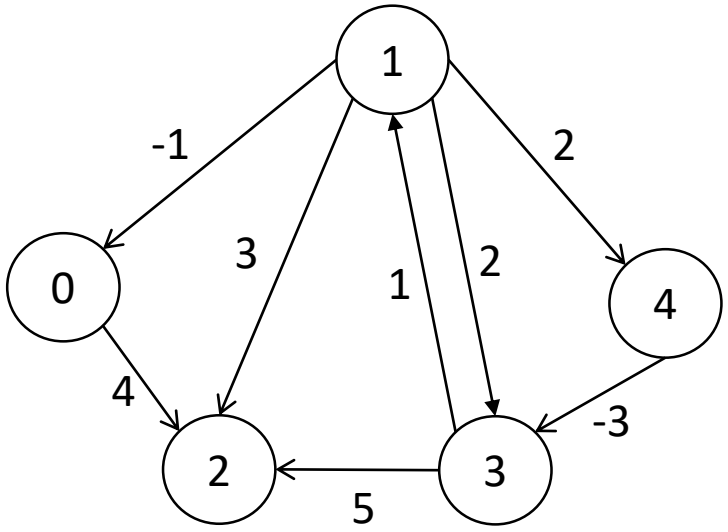
2. Consider each vertex (i) in between pair of any two vertices (s, d) and find the optimal distance between s & d considering intermediate vertex i.e. dist(s,d) = dist(s,i) + dist(i,d), if dist(s,i) + dist(i,d) < dist(s,d).

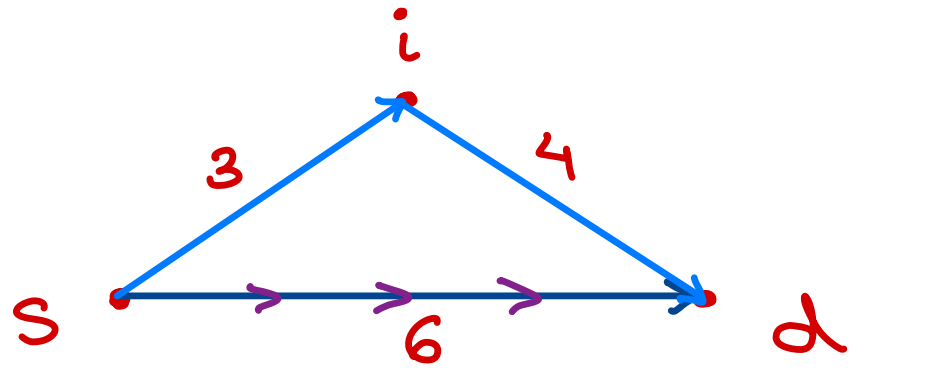
	0	1	2	3	4
0	0	∞	4	∞	∞
1	-1	0	3	2	2
2	∞	∞	0	∞	∞
3	∞	1	5	0	∞
4	∞	∞	∞	-3	0

$O(V^3)$

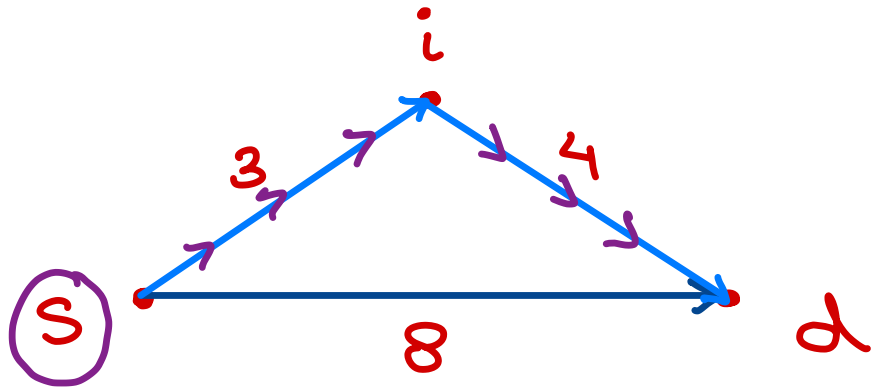


Warshall Floyd Algorithm





dist[s,d] = dist[s,d]



dist[s,d] = dist[s,i] + dist[i,d]

Johnson's Algorithm

→ all pair shortest path.

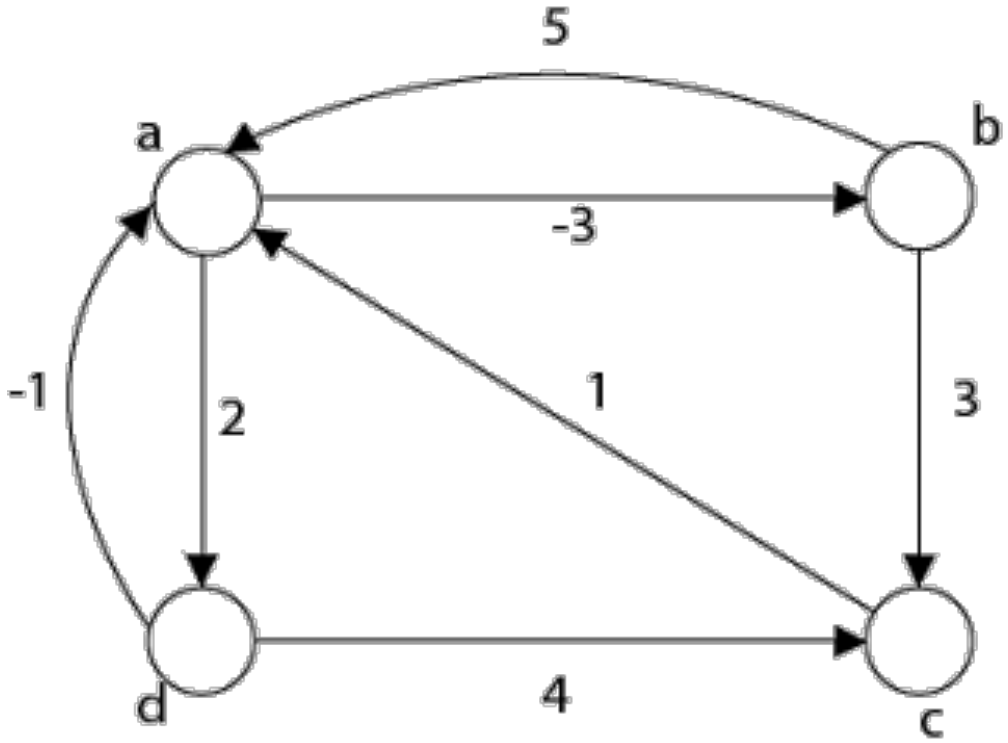
Using Bellaman for all vertices: O(V * V * E)

↳ slower than warshall floyd.

- Time complexity of Warshall Floyd is $O(V^3)$. → V * V * V
- Applying Dijkstra's algorithm on V vertices will cause time complexity O(V * V log V). This is faster than Warshall Floyd.
- However Dijkstra's algorithm can't work with –ve weight edges.
- Johnson use Bellman ford to reweight all edges in graph to remove –ve edges. Then apply Dijkstra to all vertices to calculate shortest distance. Finally reweight distance to consider original edge weights.
- Time complexity of the algorithm:

$O(VE + V^2 \log V)$. → faster than Warshall Floyd.

m

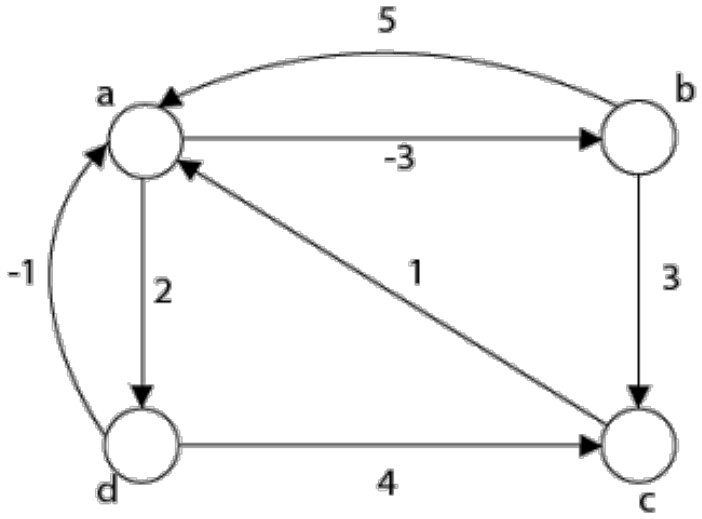


Johnson's Algorithm

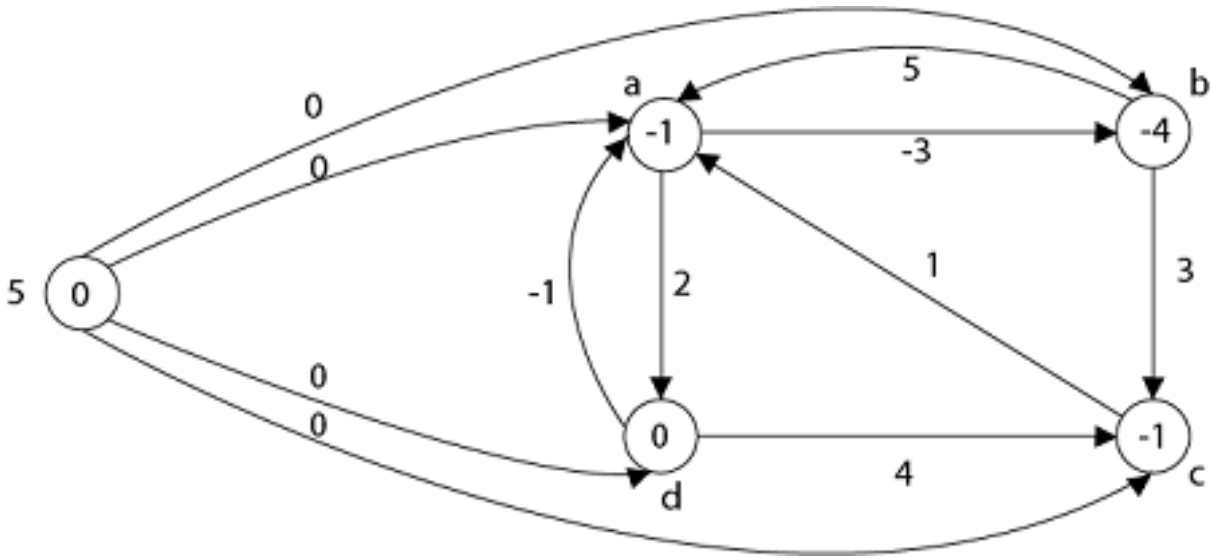
1. Add a vertex (s) into a graph and add edges from it all other vertices, with weight 0.
2. Find shortest distance of all vertices from (s) using Bellman Ford algorithm.

   a = -1, b = -4, c = -1, d = 0 and s = 0
3. Reweight all edges (u, v) in the graph, so that, they become non negative.

   weight(u, v) = weight(u, v) + d(u) – d(v)



- w(a, b) = 0
- w(b, a) = 2
- w(b, c) = 0
- w(c, a) = 1
- w(d, c) = 5
- w(d, a) = 0
- w(a, d) = 1

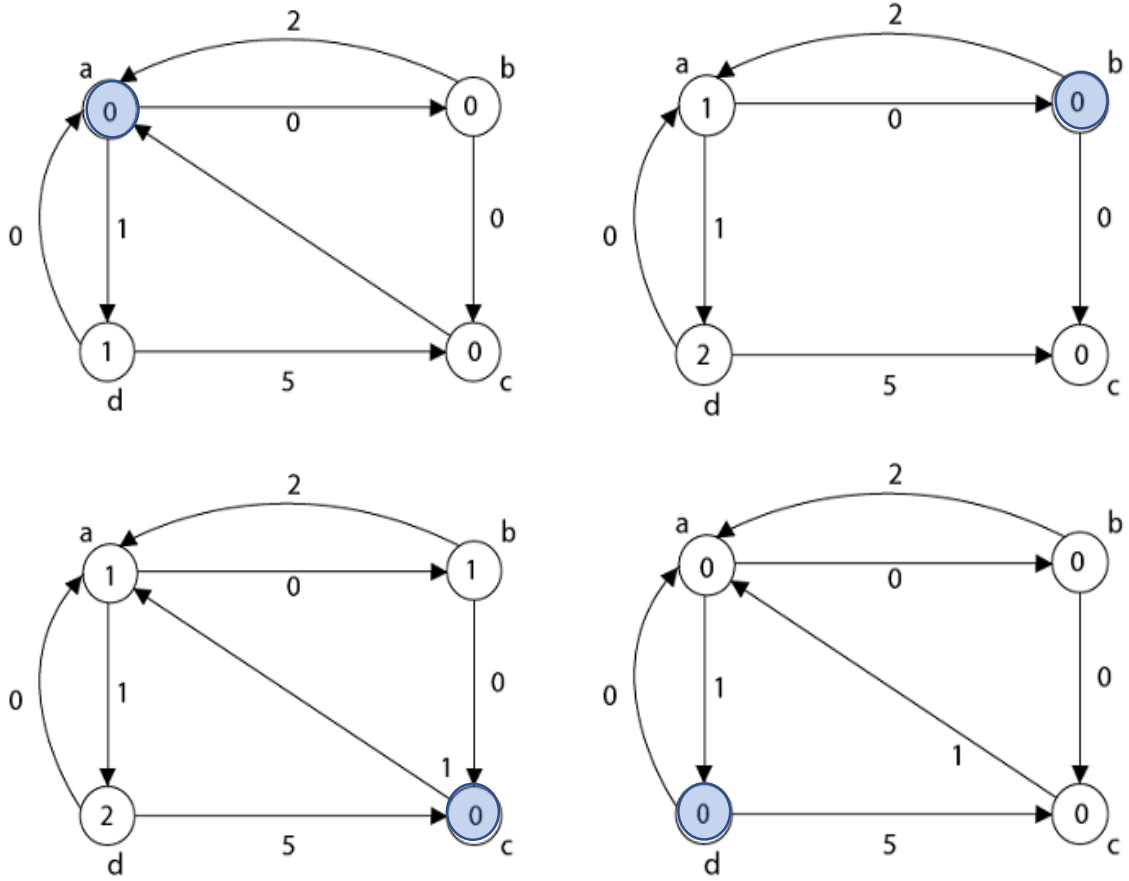


Johnson's Algorithm

4. Apply Dijkstra on each vertex to calculate shortest distance to all other vertices.
5. Reweight all distances to consider original weights.

$$dist(u, v) = dist(u, v) + d(v) - d(u)$$

	a	b	c	d
a	0 -> 0	0 -> -3	0 -> 0	1 -> 2
b	1 -> 4	0 -> 0	0 -> 3	2 -> 6
c	1 -> 1	1 -> -2	0 -> 0	2 -> 3
d	0 -> -1	0 -> -4	0 -> -1	0 -> 0

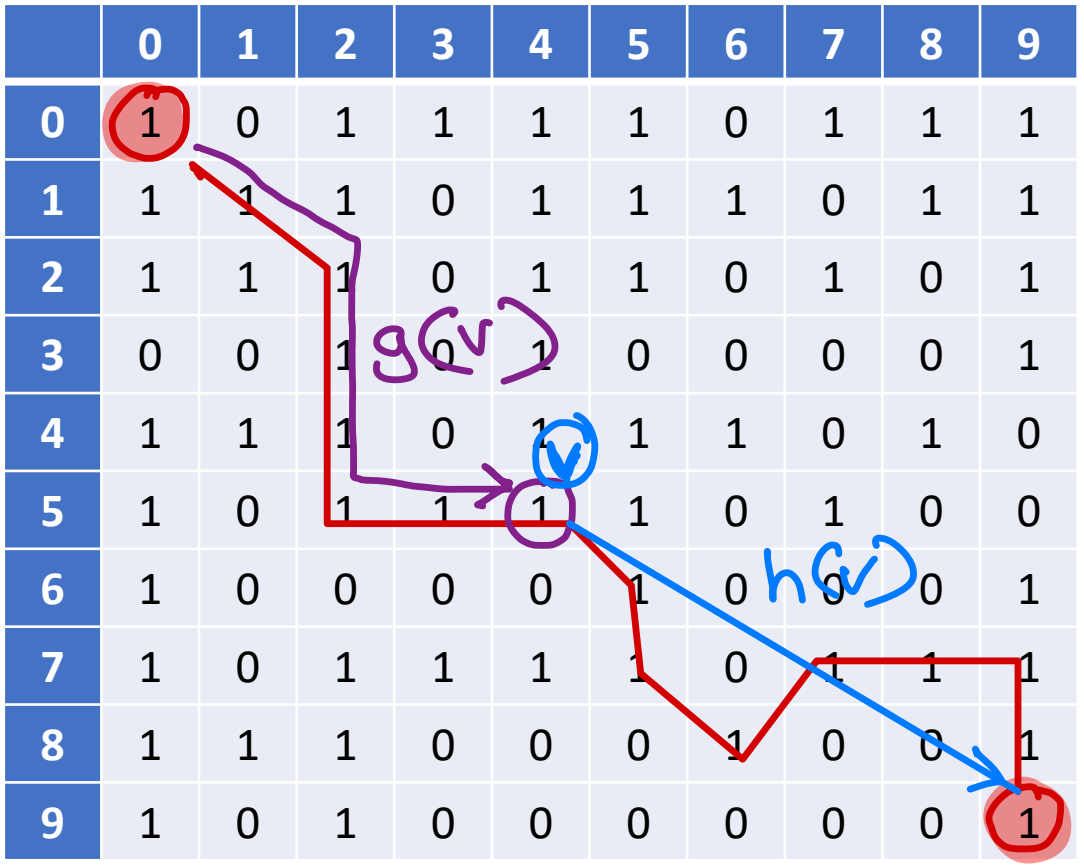


A* Search Algorithm → point to point

- Point to point approximate shortest path finder algorithm.
- This algorithm is used in artificial intelligence.
- Commonly used in games or maps to find shortest distance in faster way.
- It is modification of BFS.
- Put selected adjacent vertices on queue, based on some heuristic.
- A math function is calculated for vertices
  - f(v) = g(v) + h(v) → vertex with min f(v) is picked.
  - g(v) → cost of source to vertex v
  - *h(v)* → estimated cost of vertex v to destination.

→ Robot path Games

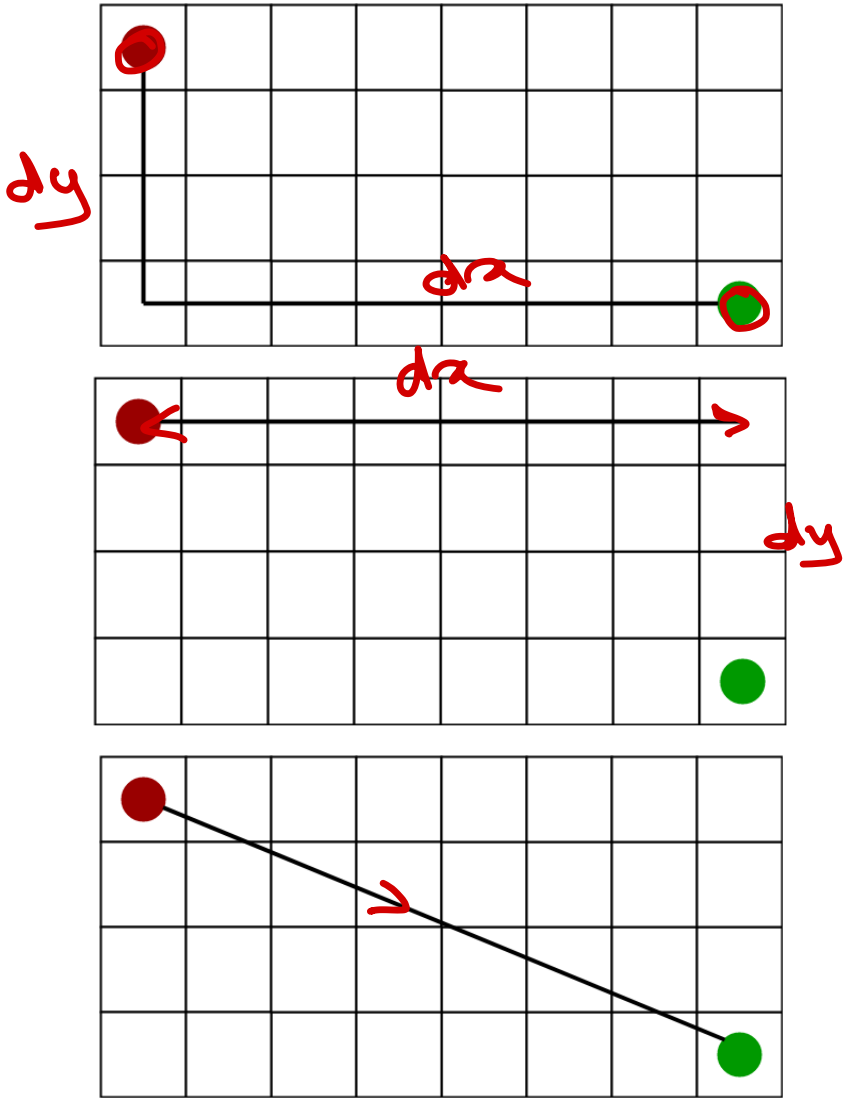
priority



	0	1	2	3	4	5	6	7	8	9
0	1	0	1	1	1	1	0	1	1	1
1	1	1	1	0	1	1	1	0	1	1
2	1	1	1	0	1	1	0	1	0	1
3	0	0	1	0	1	0	0	0	0	1
4	1	1	1	0	1	1	1	0	1	0
5	1	0	1	1	1	1	0	1	0	0
6	1	0	0	0	0	1	0	0	0	1
7	1	0	1	1	1	1	0	1	1	1
8	1	1	1	0	0	0	1	0	0	1
9	1	0	1	0	0	0	0	0	0	1

A* Search Algorithm

- h(v) represent heuristic and depends on problem domain. Three common techniques to calculate heuristic:
- Manhattan distance
  - When moves are limited in four directions only.
  - h = dx + dy
- Diagonal distance
  - When moves are allowed in all eight directions (one step).
  - h = MAX(dx, dy)
- Euclidean distance
  - When moves are allowed in any direction.
  - h = $\sqrt{dx^2 + dy^2}$
- Note that heuristic may result in longer paths in typical cases.



A* search algorithm

- Start point g(v) = 0, h(v) = 0 & f(h) = 0.
- Push start point vertex on a priority queue (by f(v)).
- Until queue is empty
  - Pop a point (v) from queue.
  - Add v into the path.
  - For each adjacent point (u)
    - If u is destination, build the path.
    - If u is invalid or already on path or blocked, skip it.
    - Calculate newg = g(v) + 1, newh = *heuristic* and newf = newg + newh.
    - If newf is less than f(u), f(u) = newf and also parent(u) = v. Rearrange elements in priority queue.

	0	1	2	3	4	5	6	7	8	9
0	1	0	1	1	1	1	0	1	1	1
1	1	1	1	0	1	1	1	0	1	1
2	1	1	1	0	1	1	0	1	0	1
3	0	0	1	0	1	0	0	0	0	1
4	1	1	1	0	1	1	1	0	1	0
5	1	0	1	1	1	1	0	1	0	0
6	1	0	0	0	0	1	0	0	0	1
7	1	0	1	1	1	1	0	1	1	1
8	1	1	1	0	0	0	1	0	0	1
9	1	0	1	0	0	0	0	0	0	1

A* Search Algorithm

	0	1	2	3	4	5	6	7	8	9
0	1	0	1	1	1	1	0	1	1	1
1	1	1	1	0	1	1	1	0	1	1
2	1	1	1	0	1	1	0	1	0	1
3	0	0	1	0	1	0	0	0	0	1
4	1	1	1	0	1	1	1	0	1	0
5	1	0	1	1	1	1	0	1	0	0
6	1	0	0	0	0	1	0	0	0	1
7	1	0	1	1	1	1	0	1	1	1
8	1	1	1	0	0	0	1	0	0	1
9	1	0	1	0	0	0	0	0	0	1

	0	1	2	3	4	5	6	7	8	9
0	1	0	1	1	1	1	0	1	1	1
1	1	1	1	0	1	1	1	0	1	1
2	1	1	1	0	1	1	0	1	0	1
3	0	0	1	0	1	0	0	0	0	1
4	1	1	1	0	1	1	1	0	1	0
5	1	0	1	1	1	1	0	1	0	0
6	1	0	0	0	0	1	0	0	0	1
7	1	0	1	1	1	1	0	1	1	1
8	1	1	1	0	0	0	1	0	0	1
9	1	0	1	0	0	0	0	0	0	1

Graph applications

- Graph represents flow of computation/tasks. It is used for resource planning and scheduling. MST algorithms are used for resource conservation. DAG are used for scheduling in Spark or Tez.

- In OS, process and resources are treated as vertices and their usage is treated as edges. This resource allocation algorithm is used to detect deadlock.

- In social networking sites, each person is a vertex and their connection is an edge. In Facebook person search or friend suggestion algorithms use graph concepts.

Thank you!

Nilesh Ghule <Nilesh@sunbeaminfo.com>