Sunbeam Institute of Information Technology

Pune and Karad

Module – Data Structures and Algorithms

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Algorithm analysis

- it is done for efficiency measurement and also known as time/space complexity
- It is done to finding time and space requirements of the algorithm
    1. Time - time required to execute the algorithm (ns, us, ms.....)
    2. Space - space required to execute the algorithm inside memory (bytes, kb, mb...—)

- finding exact time and space of the algorithm is not possible because it depends on few external factors like
    - time is dependent on type of machine (CPU), number of processes running at that time
    - space is dependent on type of machine (architecture), data types

- Approximate time and space analysis of the algorithm is always done
- Mathematical approach is used to find time and space requirements of the algorithm and it is known as "Asymptotic analysis"
- Asymptotic analysis also tells about behaviour of the algorithm for different input or for change in sequence of input
- This behaviour of the algorithm is observed in three different cases
    1. Best case
    2. Average case
    3. Worst case

'Big O' notation is used to denote space & time complexity.

Time complexity

- time is directly proportional to number of iterations of the loops used in an algorithm
- To find time complexity/requirement of the algorithm count number of iterations of the loops

1. Print 1D array on console

```
void print1DArray(int arr[]) {
    for(int i=0; i<n ; i++)
        sysout(arr[i]);
}
```

No. of loop iterations = n

Time $\propto$ n

$$\boxed{T(n) = O(n)}$$

2. Print 2D array on console

```
void print2DArray(int arr[][]) {
    for(int i=0; i< m ; i++) {
        for(int j=0 ; j<n; j++)
            sysout(arr[i][j]);
    }
}
```

Iterations of outer loop = m

Iterations of inner loop = n

Total iterations = m * n

Time $\propto$ m * n

$$\boxed{T(m,n) = O(m*n)}$$

$\because$ m == n

Time $\propto n^2$

$$\boxed{T(n) = O(n^2)}$$

Time complexity

3. Add two numbers

```
int sum(int n1, int n2) {
    int s = n1 + n2;
    return s;
}
```

- time will be constant irrespective of values of n1 & n2
- constant time requirement

$$\boxed{T(n) = O(1)}$$

4. Print table of given number

```
void printTable( int num) {
    for(int i=1; i<=10 ; i++)
        sysout ( num * i);
}
```

No. of iterations of loop = 10

- irrespective of value of num, loop will iterate constant number of times
- constant time requirement.

$$\boxed{T(n) = O(1)}$$

Time complexity

5. Print binary of decimal number

$$
\begin{array}{r|l}
2 & 9 \\ \hline
 & 4 \quad 1 \\ \hline
 & 2 \quad 0 \\ \hline
 & 1 \quad 0 \\
 & \quad\;\; 1
\end{array}
$$

$(9)_{10} = (1001)_2$

n	n>0	rem
9	T	1
4	T	0
2	T	0
1	T	1
0	F	

```
void printBinary(int n) {
    while(n>0) {
        sysout(n%2);
        n = n/2;
    }
}
```

$n = n, n/2, n/4, n/8 \ldots\ldots$

$n = \frac{n}{2^0}, \frac{n}{2^1}, \frac{n}{2^2} \ldots\ldots\ldots \frac{n}{2^{itr}}$

$\because$ for $n=1$, last time loop will be executed.

$\frac{n}{2^{itr}} = 1$

$n = 2^{itr}$

$\log 2^{itr} = \log n$

$itr \log 2 = \log n$

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

Time $\propto$ itr

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

$\boxed{T(n) = O(\log n)}$

Time complexity

Time complexities : O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), …., O(2^n), ……

Modification : + or - : time complexity is in terms of n
Modification : * or / : time complexity is in terms of log n

```
for(i=0; i<n; i++)  -> O(n)
for(i=n; i>0; i--)  -> O(n)
for (i=0; i<n; i+=20) -> O(n)
for (i=1; i<=10; i++) -> O(1)
for (i=n; i>0; i/=2) -> O(log n)
for (i=1; i<n; i*=2) -> O(log n)
```

n=9
i/=2 → 9, 4, 2, 1
i*=2 → 1, 2, 4, 8

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)   } -> O(n^2)
```

```
for(i=0; i<n; i++); -> n
                       +  = 2n     T ∝ 2n
for(j=0; j<n; j++); -> n            O(n)
```

```
for(i=0; i<n; i++) -> n
                      *            T ∝ n log n
    for(j=n; j>0; j/=2) -> log n   O(n log n)
```

Space complexity

- Finding approximate space requirement of the algorithm to execute inside memory

Total space = Input space + Auxiliary space

Input space: space required to store input

Auxiliary space: space required to process the input

Find sum of array element

```
int sumArray(int arr[], int n)
{
    sum=0;
    for(i=0; i<n; i++)
        sum += arr[i];
    return sum;
}
```

Input space → Input variables → arr

Auxiliary space → Processing variables → n, i, sum

Input space = n

Auxillary space = 3

Total space = n+3

space $\propto$ n+3

space $\propto$ n ∵ n >>>>

$\boxed{S(n) = O(n)}$

Auxiliary space → processing variables → n, i, sum

Auxiliary space $\propto$ 3

$AS(n) = O(1)$

Algorithm analysis

Iterative

- loops are used

```
int fact(int num) {
    int f=1;
  for(int i=1; i<=num; i++)
      f *= i;
  return f;
}
```

Time ∝ No. of iterations

Time $\propto n$

$T(n) = O(n)$

$AS(n) = O(1)$

Recursive

- recursion is used

```
int rfact(int num) {
  if(num==1)
     return 1;
  return num * rfact(num-1);
}
```

Time ∝ No. of recursive calls

Time $\propto n$

$T(n) = O(n)$

$AS(n) = O(n)$

Searching algorithms analysis

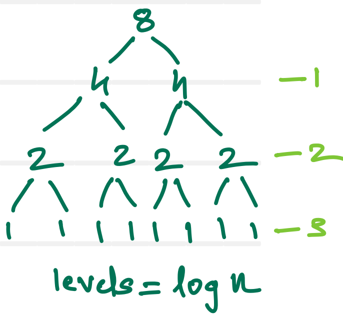
- Time is directly proportional to number of comparisons
- For searching and sorting algorithms, count number of comparisons done

1. Linear search

- Best case - Key is found at first few locations → $O(1)$
- Average case - Key is found at middle few locations → $O(n)$
- Worst case - Key is not found or found at last few locations → $O(n)$

2. Binary search

- Best case - Key is found at first few levels → $O(1)$
- Average case - Key is found at middle few levels → $O(\log n)$
- Worst case - Key is found at last level or not found → $O(\log n)$



Selection sort

1. Select one position of the array
2. Find smallest element out of remaining elements
3. Swap selected position element and smallest element
4. Repeat above steps until array is sorted (N-1)

No. of elements = n

No. of passes = n-1

n=6

pass	comp	
1	5	n-1
2	4	n-2
3	3	⋮
4	2	
5	1	1

$$\text{Total comps} = 1+2+\dots+\underbrace{(n-1)}_{n}$$

$$= \frac{n(n+1)}{2}$$

$$\text{Time} \propto \frac{1}{2}(n^2+n)$$

$$\left.\begin{array}{l}\text{Best}\\ \text{Avg}\\ \text{worst}\end{array}\right\} \boxed{T(n) = O(n^2)}$$

Mathematical Polynomial

Highest degree – degree of polynomial

– while writing time complexity consider only highest degree term because it is highest growing term in the polynomial

n	n^2
1	1
10	100
100	10000
1000	1000000

Selection sort

arr	44	11	55	22	66	33
	0	1	2	3	4	5

Pass 1

44	11	55	22	66	33
0					
11	44	55	22	66	33

Pass 2

11	44	55	22	66	33
	1				j
11	22	55	44	66	33

Pass 3

11	22	55	44	66	33
		2			
11	22	33	44	66	55

Pass 4

11	22	33	44	66	55
			3		
11	22	33	44	66	55

Pass 5

11	22	33	44	66	55
				4	
11	22	33	44	55	66

Bubble sort

1. Compare all pairs of consecutive elements of the array one by one
2. If right element is less than left element , swap both
3. Repeat above steps until array is sorted

$N = 6$

i	j ($j < N-1$)	j ($j < N-i$)
1	0 - 4	0 - 4
2	0 - 4	0 - 3
3	0 - 4	0 - 2
4	0 - 4	0 - 1
5	0 - 4	0 - 0

Best case

11 22 33 44

if array is sorted only n-1 comps will enough to check it.

$\boxed{T(n) = O(n)}$

No. of elements = n

No. of passes = n-1

$n = 6$

Pass	Comps	
1	5	n-1
2	4	n-2
3	3	⋮
4	2	⋮
5	1	1

$$\text{Total comps} = 1 + 2 + 3 \ldots\ldots + (n-1) = \frac{n(n+1)}{2}$$

n

$$\text{Time} \propto \frac{1}{2}(n^2 + n)$$

Avg, Worst: $\boxed{T(n) = O(n^2)}$

Bubble sort

33	22	66	55	44	11
0	1	2	3	4	5

33	22	66	55	44	11
22	33	66	55	44	11
22	33	66	55	44	11
22	33	55	66	44	11
22	33	55	44	66	11
22	33	55	44	11	66

22	33	55	44	11	66
22	33	55	44	11	66
22	33	55	44	11	66
22	33	44	55	11	66
22	33	44	11	55	66

22	33	44	11	55	66
22	33	44	11	55	66
22	33	44	11	55	66
22	33	11	44	55	66

22	33	11	44	55	66
22	33	11	44	55	66
22	11	33	44	55	66

22	11	33	44	55	66
11	22	33	44	55	66

Insertion sort

1. Pick one element of the array (start from 2nd index)
2. Compare picked element with all its left neighbours one by one
3. If left neighbour is greater, move it one position ahead
4. Insert picked element at its appropriate position
5. Repeat above steps until array is sorted

$n=6$

pass	comps	
1	1	1
2	2	2
3	3	3
4	4	⋮
5	5	n-1

No. of elements $= n$

No. of passes $= n-1$

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

$= \frac{n(n+1)}{2}$

$\frac{(n-1)}{n}$

Time $\propto \frac{1}{2}(n^2+n)$

Avg, Worst: $\boxed{T(n) = O(n^2)}$

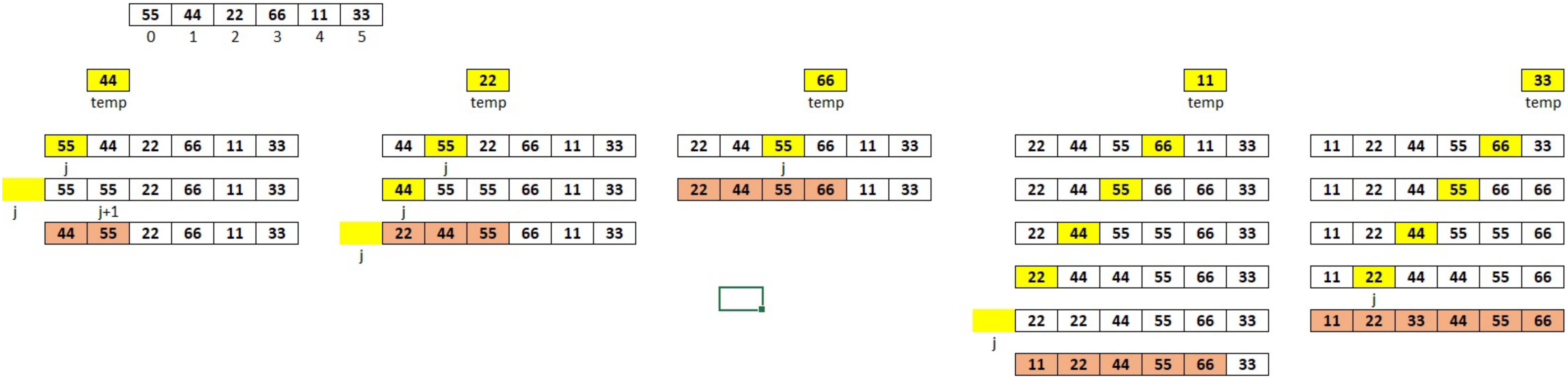
Best case

11 22 33 44

if array is sorted only n-1 comps will enough to check it.

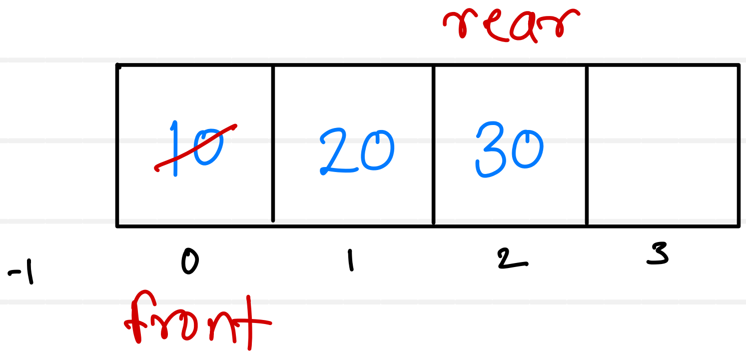
$\boxed{T(n) = O(n)}$

Insertion sort



Linear queue

- linear data structure which stores similar type of data.
- data is inserted from one end (rear)
- data is removed from another end (front)
- works on principle of "FIFO" / "First In First Out"



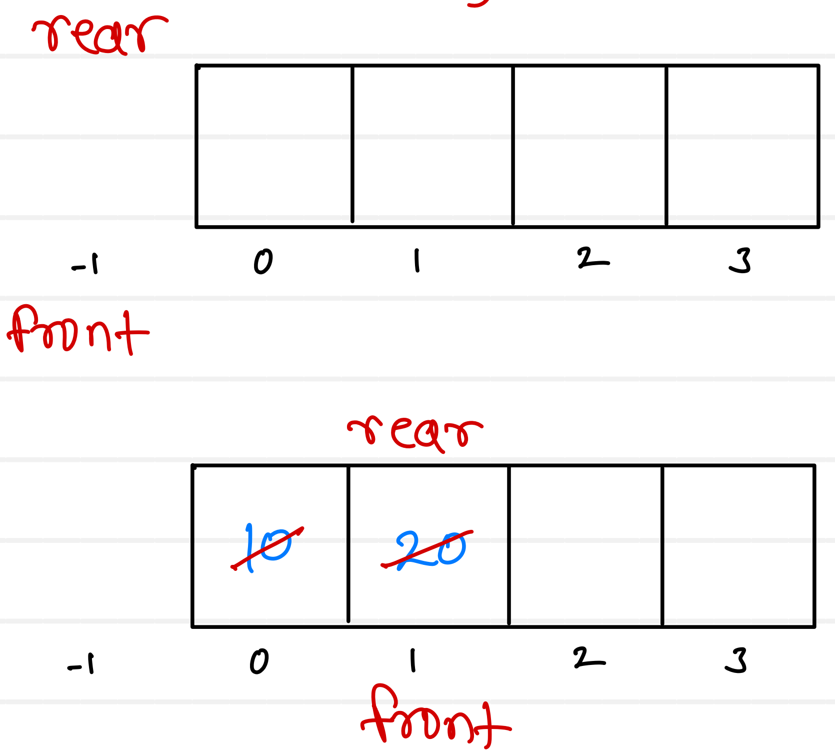
Operations:

1) Push / add / insert / Enqueue :
   - reposition rear (inc)
   - add value at rear index
2) Pop / delete / remove / Dequeue :
   - reposition front (inc)
3) Peek :
   - read / return data of front+1 index

All operations of queue are performed in O(1) time complexity.

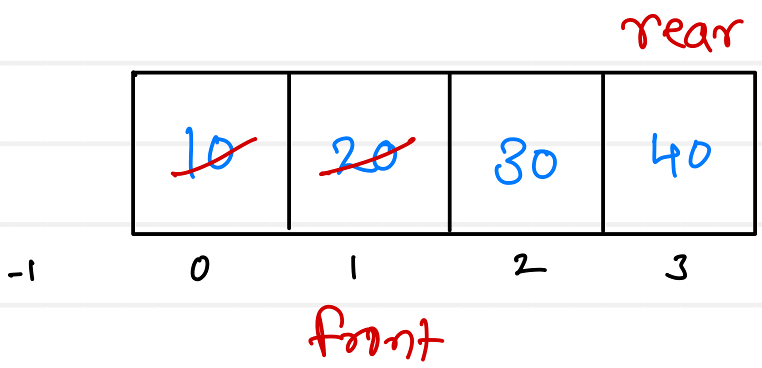
Linear queue - Conditions

Empty



front == rear

Full



rear == size-1

- if rear reaches last index of array and few initial elements are deleted, then still will not be able to use those empty locations. This will lead to poor memory utilization
- solution for this is circular queue

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com