दत्तांशरचना विधिकल्पाश्च

DATA STRUCTURE AND ALGORITHMS

OUTLINE

Data structures: Arrangement of data so that they (data items) can be used efficiently in memory.

Algorithms: Sequence of steps on data using efficient data structures to solve a given problem.

Data base: Collection of information in permanent storage for faster retrieval and updation.

Data Warehousing: Management of huge amount of legacy data for better analysis.

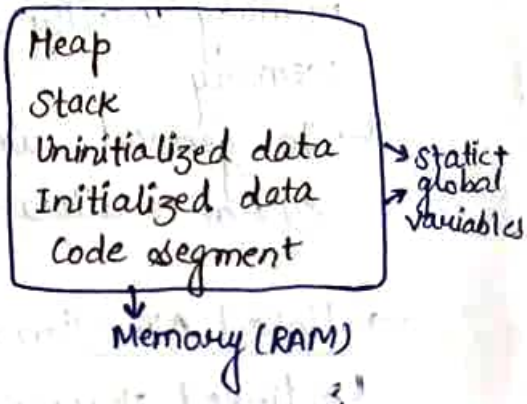
Big data: Analysis of too large or complex data which cannot be dealt with traditional data processing application.

Memory Layout of a Program

When the program starts, it's copied to the main memory.

Stack holds the memory ~~copied~~ occupied by the functions.

Heap contains the data which is requested by the program as dynamic memory.



Initialised and uninitialized data segments hold initialized and uninitialized global variables, respectively.

Memory Allocation

→ Basic forms of memory allocation:
- Contiguous
- Linked
- Indexed

→ Prototypical examples:
- Contiguous allocation: arrays
- Linked allocation: linked lists

→ Other data structures:
- Trees
- Hybrids
- Higher-demensional arrays.

→ Contiguous Allocation:

↳ An array stores n objects in a single contiguous space of memory.

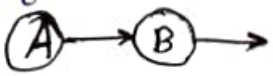
↳ Unfortunately, if more memory is required, a request for new memory usually requires copying all information into the new memory.

↳ In general, we cannot request ~~the~~ for the operating system to allocate to us the next n memory locations.

→ Linked Allocation:

↳ Linked storage such as a linked list associates two pieces of data with each item being stored:
- the object itself, and
- a reference to the next item.

In c++, that reference is the address of the next node.



Eg. This is a class describing such a node:

```
template <typename type>
class Node {
    private:
        Type node_value;
        Node * next_node;
    public:
        //...
};
```

→ The operations on this node must include:

- Constructing a new node
- Accessing (retrieving) the value
- Accessing the next pointer.

```
Node (const Type& = Type(), Node * = nullptr);
Type value() const;
Node *next const;
```

→ Pointing to nothing has been represented as:

C ——— NULL
Java ——— null
C++(old) ——— 0
C++(new) ——— nullptr
Symbolically — ϕ

→ For a linked list, however, we also require an object which links to the first object.

→ The actual linked list class must store two pointers:

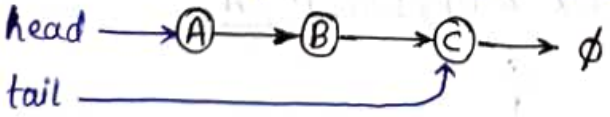
- a head and a tail:

```
Node *head;
Node * tail;
```

↳ Optionally, we can also keep a count.

```
int count;
```

↳ The next node of the last node is assigned nullptr.



→ The class structure would be:

```
template <typename Type>
class List{
    private:
        Node<Type> *head;
        Node<Type> * tail;
        int count;
    public:
        // constructor(s) ...
        // accessor(s) ...
        // mutator(s) ...
};
```

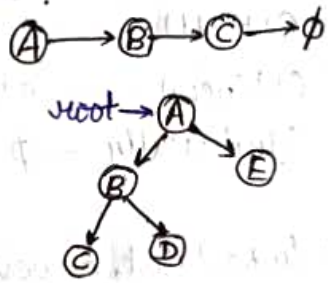
Other Allocation formats

We'll look at some variations or hybrids of these memory allocations including:

- Trees
- Graphs
- Deques (linked arrays)
- inodes

Trees:

→ The linked list can be used to store linearly ordered data.
↳ What if we have multiple 'next' pointers?



→ A rooted tree is similar to a linked list but with multiple next pointers.

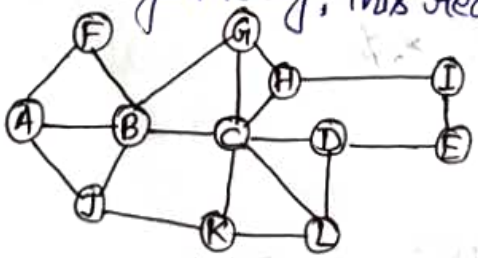
→ A tree is a variation on a linked list:
- Each node points to an arbitrary no. of subsequent nodes.
- useful for storing hierarchial data.
- Also useful for storing sorted data.

→ Usually, we will restrict ourselves to trees where each node points to at most two other nodes.

Graphs:

→ Suppose we allow arbitrary relations b/w any two objects in a container.
→ Given n objects, there are n^2-n possible relations.
↳ If we allow symmetry, this reduces to $\frac{n^2-n}{2}$.

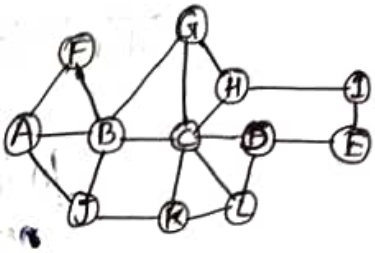
Eg



Arrays:

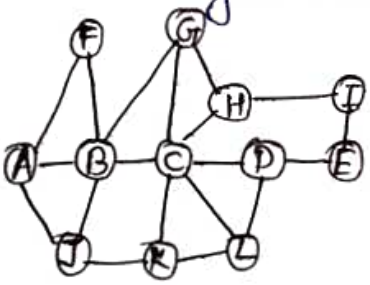
Suppose we allow arbitrary relations b/w any two objects in a container.
↳ we could represent this using a two dimensional array.

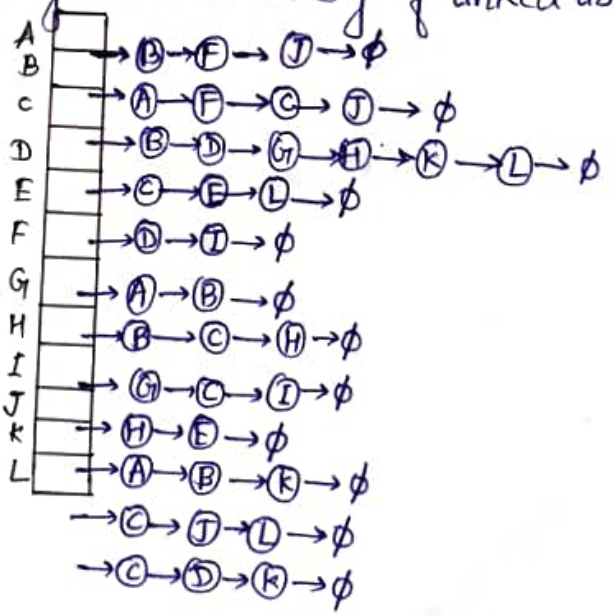
	A	B	C	D	E	F	G	H	I	J	K	L
A		X				X				X		
B	X		X			X	X			X		
C		X		X			X	X			X	X
D			X		X							X
E				X					X			
F	X	X										
G		X	X					X				
H			X				X		X			
I					X			X				
J	X	X									X	
K			X							X		X
L			X	X							X	



Arrays of Linked Lists

Suppose we allow arbitrary relations b/w any two objects in a container
↳ Alternatively, we could use a hybrid: an array of linked lists.



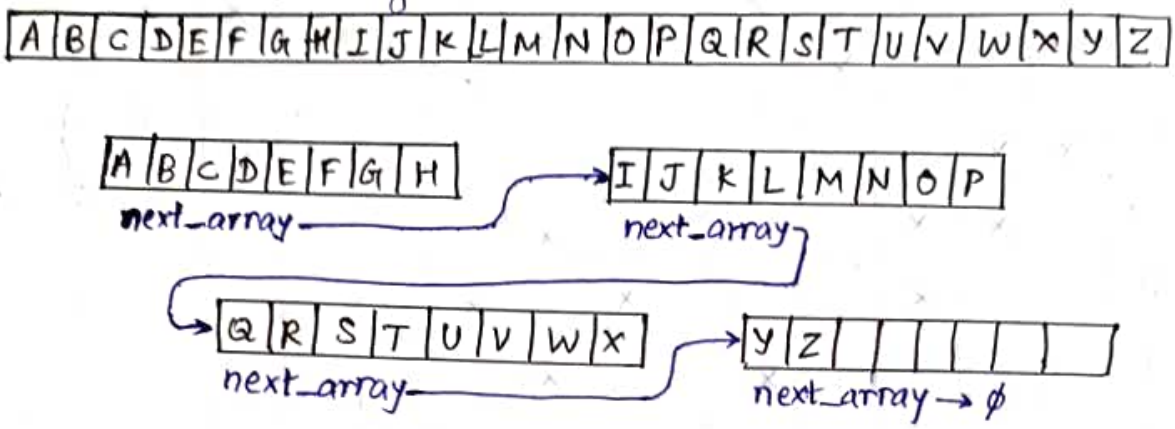


Linked Arrays:

↳ Something like this is used for the C++ STL deque container.

Eg the alphabet could be stored either as:

- an array of 26 entries, or
- a linked list of arrays of 8 entries.



Algorithm run times:

Once we have chosen a data structure to store both the objects and the relationships, we must implement the queries or operations as algorithms.

- The Abstract Data Type will be implemented as a class.
- The data structure will be defined by the member variables.
- The member functions will implement the algorithms.

ASYMPTOTIC ANALYSIS

Suppose we have two algorithms, how can we tell which is better?

↳ we could implement both algorithms, run them both
  ↳ Expensive and error prone.

↳ Preferably, we should analyze them mathematically
  ↳ Algorithm analysis.

Maximum Value

For example, the time taken to find the largest object in an array of n random integers will take n operations.

```
int find_max(int *array, int n) {
    int max = array[0];

    for(int i=1; i<n; ++i) {
        if(array[i] > max){
            max = array[i];
        }
    }

    return max;
}
```

→ One comment:

- In this class, we will look at both simple C++ arrays and the standard template library (STL) structures.
- Instead of using the built-in array, we could use the STL vector class
- The vector class is closer to the C#/Java array.

```
#include <vector>
int find_max(std::vector<int> array){
    if(array.size() == 0){
        throw underflow();
    }
    int max = array[0];
```

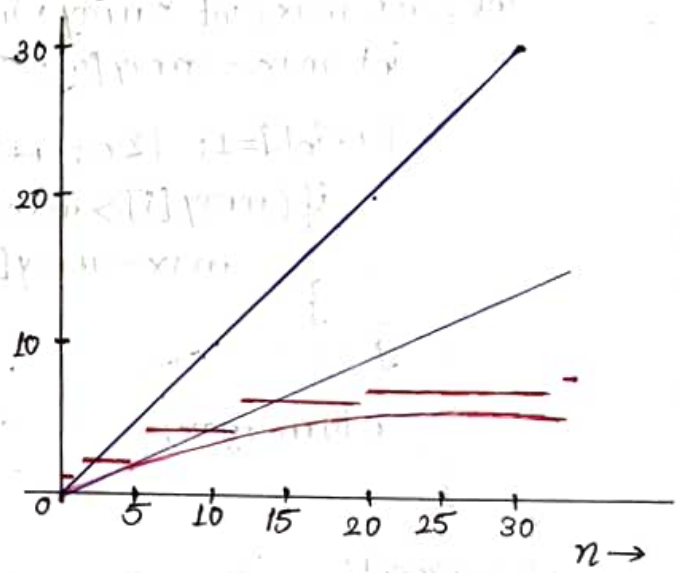
```
    for (int i=1; i< array.size(); ++i){
        if (array[i]> max){
            max = array[i];
        }
    }
    return max;
}
```

Linear and Binary Search

There are other algorithms which are significantly faster as the problem size increases.

The plot shows maximum and average no. of comparisons to find ~~the~~ an entry in a sorted array of size n.

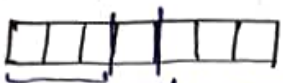
- Linear search
- Binary search



Linear search:

```
int linear_search (int value, int *array, int n) ⇝ Array is not sorted
{
  for (int i=0; i<n; ++i)
  {
     if(array[i] == value)
        return i;
  }
  return -1;
}
```

Binary Search:



value < array [n/2]

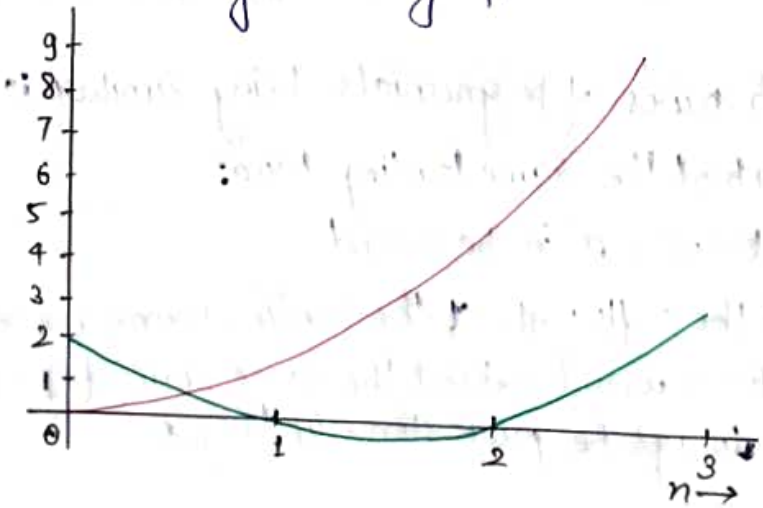
```
int binary_search (int value; int* array; int n)
{
    int a=0;
    int c=n-1;
    int b;
    while( a<=c)
    {
       b = (a+c)/2;
       if( array[b]==value)
          return b;
       else if (array[b] < value)
           a= b+1;
        else
            c= b-1;
    }
    return -1;
}
```

Quadratic Growth

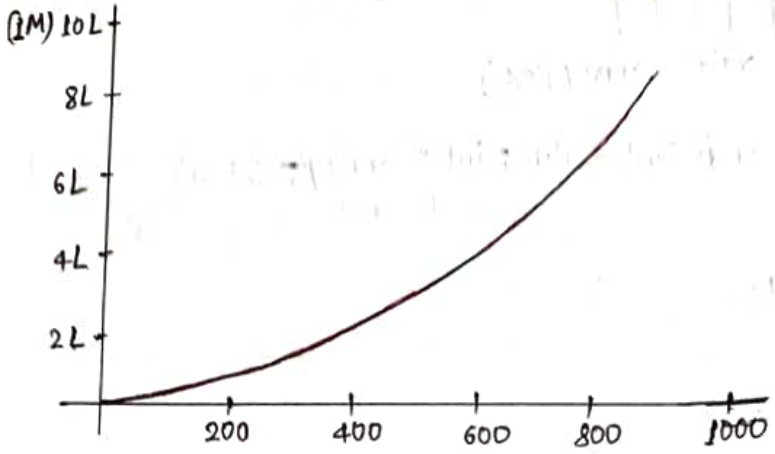
Consider the two functions

$f(n)=n^2$ and $g(n)=n^2-3n+2$.

Around n=0, they look very different:



Yet on the range $n=[0,1000]$, they are (relatively) indistinguishable:



The absolute difference is large, e.g.,

$f(1000) = 10\,00\,000$

$g(1000)$ 9 97 002

but the relative difference is very small,

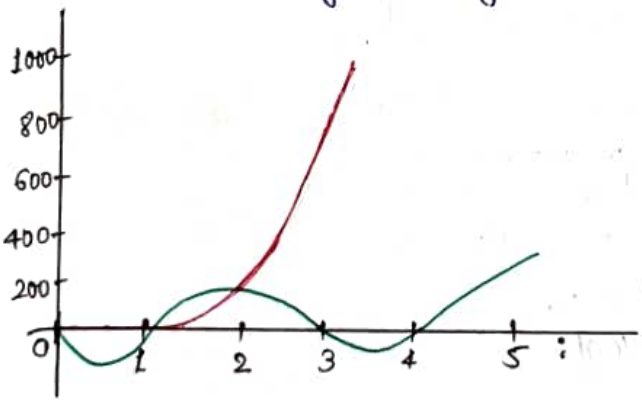
$$\left|\frac{f(1000)-g(1000)}{f(1000}\right| = 0.002998 < 0.3\%.$$

and this difference goes to zero as $n \to \infty$.

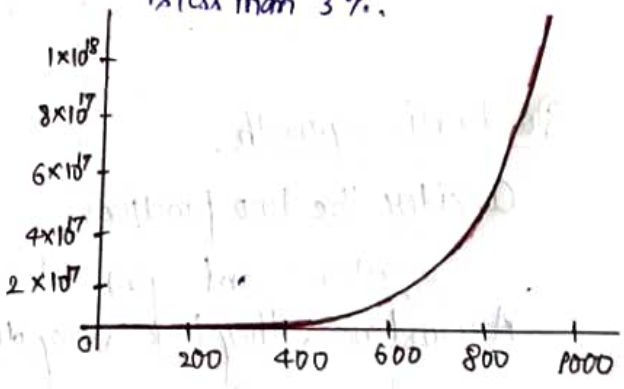
Another example:

$f(n)=n^6$ and $g(n)= n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$

Around $n=0$, they ~~took~~ are very different,



Still around $n=1000$, the relative diff. is less than 3%.



The justification for both pairs of polynomials being similar is that, in both cases, they each had the same leading term:

n^2 in the first case, n^6 in the second.

Suppose, however, that the coefficients of the leading terms were different.
↳ In this case, both functions would exhibit the same rate of growth, however, one would always be proportionally larger.

Eg. We'll look at two egs:

- A comparison of selection sort and bubble sort.
- A comparison of insertion sort and quick sort.

Asymptotic / Bachmann-Landau Notations

Time complexity: The study of the efficiency of algorithms.
↳ It tells us how much time is taken by an algorithm to process a given input.
(How time taken to execute an algorithm grows with the size of the input.)

Calculating order in terms of input size:

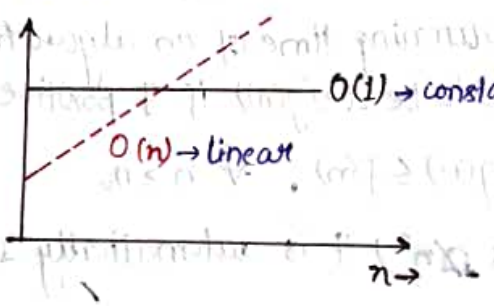
In order to calculate the order, the most impactful term containing n is taken into account. (n: size of input)

Let us assume that formula of an algorithm in terms of input size 'n' looks like this:

Algo. 1 : $\underbrace{K_1 n^2}_{\text{Highest order term}} + \underbrace{K_2 n + 36}_{\text{can ignore lower order terms}} \Rightarrow O(n^2)$

Algo. 2 : $K_1 K_2^2 + K_3 K_2 + 8$
↳ $K_1 K_2^2 n^0 + K_3 K_2 + 8 \Rightarrow O(n^0)$ or $O(1)$:

Visualising Big O:



→ Asymptotic Notations give an idea about how good a given algorithm is compared to some other algorithm.

→ We will at times use five possible descriptions:

Notation	Name	Description (asymptotically)	Limit definition	Formal definition:
① $f(n) = o(g(n))$	Small oh	f is dominated by g	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$	$\exists\, c>0, n_0\ \forall\, n \geq n_0$: $0 \leq \lvert f(n) \rvert \leq c\,g(n)$
② $f(n) = O(g(n))$	Big Oh	$\lvert f \rvert$ is bounded above by g (upto const. factor)	$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$	$\exists\, c>0, n_0\ \forall\, n \geq n_0$: $0 \leq f(n) \leq c\,g(n)$
③ $f(n) = \Theta(g(n))$	Big Theta	f is bound both above and below by g	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$	$\exists\, c_1, c_2 > 0, n_0\ \forall\, n \geq n_0$: $0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n)$
④ $f(n) = \Omega(g(n))$	Big Omega	f is bounded below by g	$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$	$\exists\, c>0, n_0\ \forall\, n \geq n_0$: $0 \leq c\,g(n) \leq f(n)$
⑤ $f(n) = \omega(g(n))$	Small Omega	f dominates g	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$	$\exists\, c>0, n_0\ \forall\, n \geq n_0$: $0 \leq c\,g(n) < f(n)$

Big Oh Notation

↳ Used to describe asymptotic upper bound.

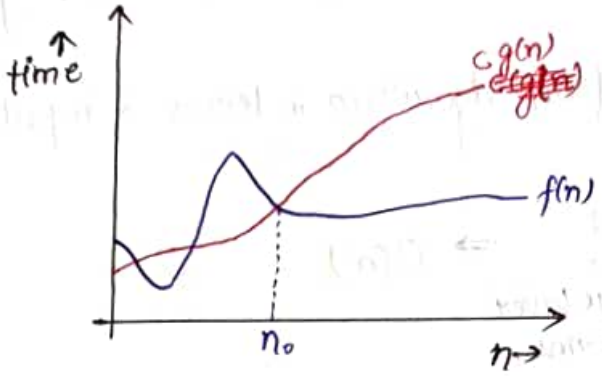
↳ Mathematically, if $f(n)$ describes running time of an algorithm, $f(n)$ is $O(g(n))$ iff there exists positive constant 'c' and 'n_0' such that

$$0 \leq f(n) \leq c\,g(n) \quad \text{for all } n \geq n_0$$

↳ used to describe upper bound on a function

↳ If a function is $O(n)$, it is automatically $O(n^2)$ as well!

Eg.



Big Omega Notation

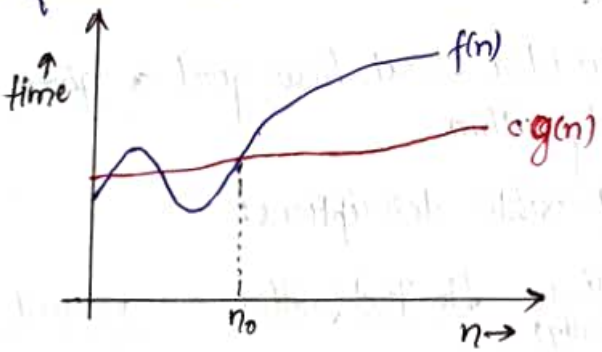
↳ Provides asymptotic lower bound.

↳ Let $f(n)$ defines running time of an algorithm.
Then $f(n)$ is said to be $\Omega(g(n))$ if $\exists$ positive constant 'c' and n_0 such that

$$0 \leq cg(n) \leq f(n), \quad \forall\ n \geq n_0$$

↳ If a function is $\Omega(n^2)$, it is automatically $\Omega(n)$ as well.

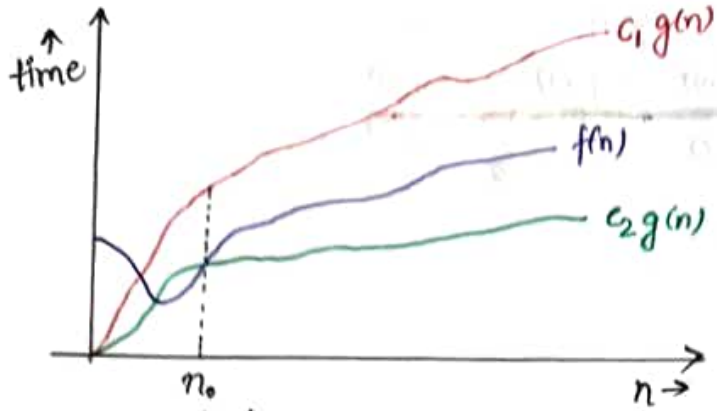
Eg.



Big Theta Notation

↳ Let $f(n)$ defines running time of an algorithm.
Then $f(n)$ is said to be $\Theta(g(n))$ iff $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.
Mathematically, there exists positive constants c_1 and c_2 such that $f(n)$ is sandwiched between $c_2\,g(n)$ and $c_1\,g(n)$.

$$0 \leq f(n) \leq c_1 g(n), \quad \forall\ n \geq n_0 \rightarrow \text{sufficiently large}$$
$$0 \leq c_2 g(n) \leq f(n) \quad \forall\ n \geq n_0 \rightarrow \text{value of } n$$
$$\Rightarrow 0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n) \quad \forall\ n \geq n_0.$$



Big Θ as an Equivalence Relation:

Eg all of

n^2, $100\,000n^2 - 4n + 19$, $n^2 + 1\,000\,000$, $323n^2 - 4n\ln(n) + 43n + 10$,

$42n^2 + 32$, $n^2 + 61n\ln^2(n) + 7n + 14\ln^3(n) + \ln(n)$

are big-Θ of each other.

↳ Eg $42n^2 + 32 = \Theta(323n^2 - 4n\ln(n) + 43n + 10)$

→ The most common classes are given names:

- $\Theta(1)$ ——— constant
- $\Theta(\ln(n))$ ——— logarithmic
- $\Theta(n)$ ——— linear
- $\Theta(n\ln(n))$ ——— "n log n"
- $\Theta(n^2)$ ——— quadratic
- $\Theta(n^3)$ ——— cubic
- $2^n, e^n, 4^n, \ldots$ ——— exponential

→ For the functions we are interested in, it can be said that

$f(n) = O(g(n))$ is equivalent to $f(n) = \Theta(g(n))$ or $f(n) = o(g(n))$

and,

$f(n) = \Omega(g(n))$ is equivalent to $f(n) = \Theta g(n)$ or $f(n) = \omega(g(n))$

Note: Big theta gives a better picture of runtime for a given algorithm.

→ It is expected to give an answer in terms of Big theta, when they ask 'order of'.

→ Increasing order of common runtimes:

$$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n^n$$

(1: better; n^n: worse)

Common runtimes from better to worse ⟶

Summary:

$O(g(n))$ $\Omega(g(n))$

We say $f(n) =$ $o(g(n))$ $\Theta(g(n))$ $\omega(g(n))$

if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ $\quad 0<c<\infty$ $\quad \infty$

Eg. $n^p = \Theta(n^q) \longrightarrow p=q$

$n^p = O(n^q) \longrightarrow p \le q$

$n^p = o(n^q) \longrightarrow p<q$

$n^p = \Omega(n^q) \longrightarrow p \ge q$

$n^p = \omega(n^q) \longrightarrow p>q$

SORTING ALGORITHMS

The Task

Given an array $x[0], x[1], \ldots, x[size-1]$

Reorder entries so that

$$x[0] <= x[1] <= \ldots <= x[size-1]$$

Here, list is in non-decreasing order.
We can also sort a list of elements in non-increasing order.

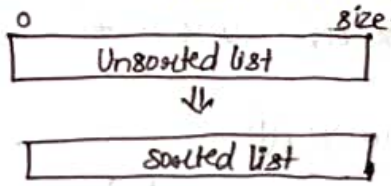
Eg. Original list: 10, 30, 20, 80, 70, 10, 60, 40, 70

Sorted in non-decr. order: 10, 10, 20, 30, 40, 60, 70, 70, 80

Sorted in non-incr. order: 80, 70, 70, 60, 40, 30, 20, 10, 10

Problem:

What do we want? → Data to be sorted in order.



Issues in Sorting

- How to rearrange a given set of data?
- Which data structures are more suitable to store data prior to their sorting?
- How fast the sorting can be achieved?
- How sorting can be done in a memory constraint situation?
- How to sort various types of data?

Sorting by Comparison

↳ Basic operation involved in this type of sorting technique is comparison.
↳ A data is compared with other items in the list of items in order to find its place in the sorted list.

- Insertion
- Selection
- Exchange
- Enumeration

Sorting by Comparison – Insertion:

↳ From a given list of items, one item is considered at a time. The item chosen is then inserted into an appropriate position relative to the previously sorted items. The item can be inserted into the same list or to a different list.

eg., Insertion sort.

Sorting by Comparison – Selection:

↳ First the smallest (or largest) item is located and it is separated from the rest; then the next smallest (or next largest) is selected and so on until all items are separated.

eg., Selection sort, Heap sort.

Sorting by comparison – Exchange:

↳ If two items are found to be out of order, they are interchanged. The process is repeated until no more exchange is required.

eg., Bubble sort, Shell sort, Quick sort.

Sorting by comparison – Enumeration:

↳ Two or more input lists are merged into an output list and while merging the items, an output list is chosen following the required sorting order.

eg., Merge sort.

Sorting by Distribution

↳ No key comparison takes place.

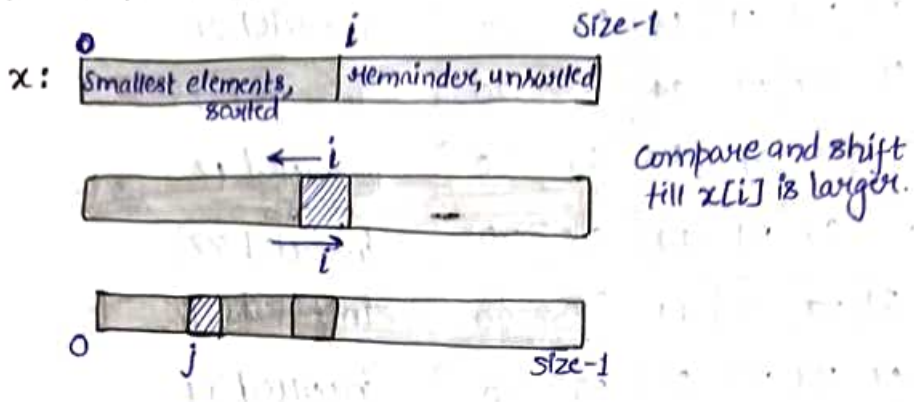
↳ All items under sorting are distributed over an auxiliary storage space based on the constituent element in each and then grouped them together to get the sorted list.

↳ Distributions of items based on the following choices:

- Radix: An item is placed in a space decided by the bases (or radix) of its components with which it is composed of.
- Counting: Items are sorted based on their relative counts.
- Hashing: Items are hashed, that is, dispersed into a list on a hash function.

Insertion Sort

General situation:




```
void insertionSort (int list[], int size)
{
    int i,j, item;

    for (i= 1; i<size; i++)
    {
       item = list[i];
/* Move elements of list[0...i-1], that are greater than item, to one
  position ahead of their current position */
        for (j=i-1; (j>=0) && (list [j] > item); j--)
              list [j+1] = list [j];
        list[j+1] = item ;
    }
}

int main()
{
   int x[ ]={-45, -89, -65, 87, 0, 3, -23, 19, 56, 21, 76,-50};
   int i;
   for(i=0; i<12; i++)
      printf ("%d", x[i]);
   printf ("\n");

   InsertionSort (x, 12);

   for(i=0; i<12; i++)
      printf ("%d", x[i]);
   printf ("\n");
}
```

Output :

```
-45 89 -65 87 0 3 -23 19 56 21 76
-65 -50 -45 -23 0 3 19 21 56 76 87 89
```

8.

54	26	93	17	77	31	44	55	20
26	54	93	17	77	31	44	55	20
26	54	93	17	77	31	44	55	20
17	26	54	93	77	31	44	55	20
17	26	54	77	93	31	44	55	20
17	26	31	54	77	93	44	55	20
17	26	31	44	54	77	93	55	20
17	26	31	44	54	55	77	93	20
17	20	26	31	44	54	55	77	93

Assume 54 is a sorted list of 1 item
inserted 26
inserted 93
inserted 17
inserted 77
Inserted 31
inserted 44
inserted 55
inserted 20

Insertion Sort: Complexity Analysis

Case-1: If the input list is already in sorted order

No. of comparisons: No. of comparison in each iteration is 1.

$C(n) = 1+1+1+\ldots+1$ upto $(n-1)^{th}$ iteration.

No. of movement: No data movement takes place in any iteration.

$M(n) = 0$

Case-2: If the input list is sorted but in reverse order

No. of comparisons: i in each iteration

$$C(n) = 1+2+3+\ldots+(n-1) = \frac{n(n-1)}{2}$$

No. of movement: i in any i^{th} iteration.

$$M(n) = 1+2+3+\ldots+(n-1) = \frac{n(n-1)}{2}$$

Case-3: If the input list is in random order

Let p_j be the probability that the key will go to the j^{th} location $(i \le j \le i+1)$. Then the no. of comparisons will be $j \cdot p_j$.

- The average no. of comparisons in the $(i+1)^{th}$ iteration is

$$A_{i+1} = \sum_{j=1}^{i+1} j \cdot p_j$$

- Assume that all keys are distinct and all permutations of keys are equally likely.

$$p_1 = p_2 = p_3 = \ldots = p_{i+1} = \frac{1}{i+1}$$

No. of comparisons: Av. no. of comparisons in the $(i+1)^{th}$ iteration,

$$A_{i+1} = \frac{1}{i+1} \sum_{j=1}^{i+1} j = \frac{1}{i+1} \cdot \frac{(i+1)\cdot(i+2)}{2} = \frac{i+2}{2}$$

Total no. of comparisons for all $(n-1)$ iterations is

$$C(n) = \sum_{i=0}^{n-1} A_{i+1} = \frac{1}{2} \cdot \frac{n(n-1)}{2} + (n-1)$$

No. of movements: Average no. of movements in the i^{th} iteration,

$$M_i = \frac{1+(i-1)+(i-2)+\ldots+2+1}{i} = \frac{i+1}{2}$$

Total no. of movements,

$$M(n) = \sum_{i=1}^{n-1} M_i = \frac{1}{2} \cdot \frac{n(n-1)}{2} + \frac{n-1}{2}$$

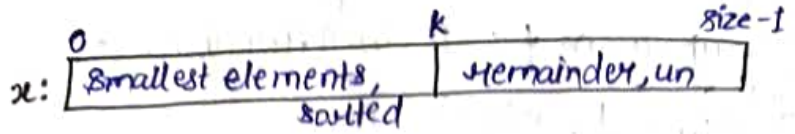
Insertion Sort: Summary of Complexity Analysis

Case	Comparisons	Movement	Memory	Remarks
Case 1	$C(n) = (n-1)$	$M(n) = 0$	$S(n) = n$	Input list is in sorted order.
case 2	$C(n) = \frac{n(n-1)}{2}$	$M(n) = \frac{n(n-1)}{2}$	$S(n) = n$	Input list is sorted in reverse order
case 3	$C(n) = \frac{(n-1)(n+4)}{4}$	$M(n) = \frac{(n-1)(n+2)}{4}$	$S(n) = n$	Input list is in random order.

Case	Run Time, $T(n)$	Complexity	Remarks
Case 1	$T(n) = c(n-1)$	$T(n) = O(n)$	Best Case
Case 2	$T(n) = cn(n-1)$	$T(n) = O(n^2)$	Worst case
Case 3	$T(n) = c\frac{(n-1)(n+3)}{2}$	$T(n) = O(n^2)$	Average Case

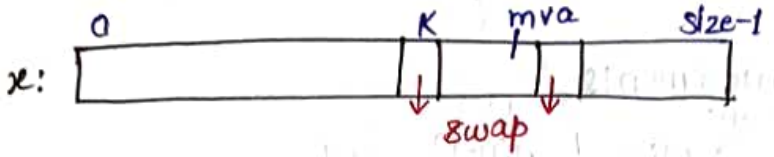
Selection Sort

General situation:



Steps:

- Find smallest element, mval, in x[k... size-1]
- Swap smallest element with x[k], then increase k.




```
/* Yield location of smallest element in x[k... size-1]; */
int findMinLoc (int x[], int k, int size)
{
  int j, pos;     //x[pos] is the smallest element found so far
  pos = k;
  for (j=k+1; j<size; j++){
     if (x[j] < x[pos])
        pos=j;
  }
  return pos;
}

/* The main sorting function
 * sort x[0...size-1] in non-decreasing order */
int selectionSort (int x[], int size)
{
  int k, m;
   for(k=0; k<size-1; k++)
  {
     m=findMinLoc (x, k, size)
     temp = a[k];
     a[k]= a[m];
     a[m] = temp;
  }
}
```

Eg.

x:	3	12	-5	6	142	21	-17	45
x:	-17	12	-5	6	142	21	3	45
x:	-17	-5	12	6	142	21	3	45
x:	-17	-5	3	6	142	21	12	45
x:	-17	-5	3	6	142	21	12	45
x:	-17	-5	3	6	12	21	142	45
x:	-17	-5	3	6	12	21	142	45
x:	-17	-5	3	6	12	21	45	142
x:	-17	-5	3	6	12	21	45	142

Selection Sort: Complexity Analysis

Case-1: If the input list is already in sorted order

No. of comparisons:

$$C(n) = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

No. of movement: No data movement takes place in any iteration

$$M(n) = 0$$

Case-2: If the input list is sorted but in reverse order

No. of comparisons:

$$c(n) = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

No. of movements:

$$M(n) = \frac{3}{2}(n-1)$$

Case-3: If the input list is in random order

No. of comparisons:

$$C(n) = \frac{n(n-1)}{2}$$

- Let p_i be the probability that the i^{th} smallest element is in the i^{th} position. No. of total swap operations $= (1-p_i) \times (n-1)$
  where $p_1 = p_2 = p_3 = \dots = p_n = \frac{1}{n}$.
- Total no. of movements,

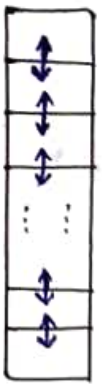
$$M(n) = \left(1-\frac{1}{n}\right) \times (n-1) \times 3 = \frac{3(n-1)(n-1)}{n}$$

Selection Sort: Summary of complexity Analysis

Case	Comparisons	Movement	Memory	Remarks
Case 1	$C(n) = \frac{n(n-1)}{2}$	$M(n) = 0$	$S(n) = 0$	Input list is in sorted order
Case 2	$C(n) = \frac{n(n-1)}{2}$	$M(n) = \frac{3(n-1)}{2}$	$S(n) = 0$	Input list is in reverse order
Case 3	$C(n) = \frac{n(n-1)}{2}$	$M(n) = \frac{3(n-1)^2}{n}$	$S(n) = 0$	Input list is in random order

Case	Rum Time, $T(n)$	Complexity	Remarks
Case 1	$T(n) = \frac{n(n-1)}{2}$	$T(n) = O(n^2)$	Best case
Case 2	$T(n) = \frac{(n-1)(n+3)}{2}$	$T(n) = O(n^2)$	Worst case
Case 3	$T(n) \approx \frac{(n-1)(2n+3)}{2}$ (Taking $n-1 \approx n$)	$T(n) = O(n^2)$	Average case

Bubble Sort



In every iteration heaviest element drops at the bottom

The bottom moves upward

The sorting process proceeds in several passes.

- In every pass, we go on comparing neighbouring pairs, and swap them out of order.
- In every pass, the largest of the elements under considering will bubble to the top (i.e., the right).

How the passes proceed?

- In pass 1, we consider index 0 to n-1.
- In pass 2, we consider index 0 to n-2.
- In pass 3, we consider index 0 to n-3.
- ...
- ...
- In pass n-1, we consider index 0 to 1.

Eg:

Pass: 1

x: 3 12 -5 6 72 21 -7 45

x: 3 12 -5 6 72 21 -7 45

x: 3 -5 12 6 72 21 -7 45

x: 3 -5 6 12 72 21 -7 45

x: 3 -5 6 12 72 21 -7 45

x: 3 -5 6 12 21 72 -7 45

x: 3 -5 6 12 21 -7 72 45

x: 3 -5 6 12 21 -7 45 72

Pass: 2

x: 3 -5 6 12 21 -7 45 72

-5 3 6 12 21 -7 45 72

-5 3 6 12 21 -7 45 72

-5 3 6 12 21 -7 45 72

-5 3 6 12 21 -7 45 72

-5 3 6 12 21 -7 45 72

-5 3 6 12 21 -7 45 72

```
void swap (int *x, int *y)
{
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

```
void bubble_sort (int x[], int n)
{
    int i,j;
    for (i= n-1; i>0; i--){
        for (j=0 ; j<i; j++){
            if (x[j] > x[j+1])
                swap (&x[j], &x[j+1]);
        }
    }
}

int main()
{
  int x[]={ -45, 89, -65, 87, 0, 3, -23, 19, 56, 21, 76, -50};
  int i;
  for(i=0; i<12; i++)
       printf ("%d", x[i]);
   printf ("\n");
   bubble_sort (x, 12);
   for (i=0; i<12; i++)
      printf ("%d", x[i]);
   printf ("\n");
}
```

Output:

```
-45  89  -65  87   0  3  -23  19  56  21  76  -50
-65  -50 -45  -23  0  3  19   21  56  76  87  89
```

Bubble Sort: Complexity Analysis

Case-1: If the input list is already in sorted order

No: of comparisons:

$$C(n) = \frac{n(n-1)}{2}$$

No: of movements:

$$M(n) = 0$$

Case-2: If the input list is sorted but in reverse order.

No. of comparisons:

$$c(n) = \frac{n(n-1)}{2}$$

No. of movements:

$$M(n) = \frac{n(n-1)}{2}$$

Case-3: If the input list is in random order.

No. of comparisons:

$$C(n) = \frac{n(n-1)}{2}$$

No. of movements:

- Let p_j be the probability that the largest element is in the unsorted part is in j^{th} $(1 \le j \le n-i+1)$ location.
- The average no. of swaps in the i^{th} pass is

$$= \sum_{j=1}^{n-i+1} (\overline{n-i+1} - j) \cdot p_j$$

- $p_1 = p_2 = p_{n-i+1} = \frac{1}{n-i+1}$
- Average no. of swaps in the i^{th} pass is

$$= \sum_{j=1}^{n-i+1} \frac{1}{n-i+1} \cdot (\overline{n-i+1} - j) = \frac{n-i}{2}$$

- The average no. of movements,

$$M(n) = \sum_{i=1}^{n-1} \frac{n-i}{2} = \frac{n(n-1)}{4}.$$

Bubble sort: Summary of Complexity Analysis

Case	Comparisons	Movement	Memory	Remarks
Case 1	$c(n) = \frac{n(n-1)}{2}$	$M(n) = 0$	$S(n) = 0$	Input list is in sorted order.
Case 2	$c(n) = \frac{n(n-1)}{2}$	$M(n) = \frac{n(n-1)}{2}$	$S(n) = 0$	Input list is sorted in reverse order.
Case 3	$c(n) = \frac{n(n-1)}{2}$	$M(n) = \frac{n(n-1)}{4}$	$S(n) = 0$	Input list is in random order.

Case	Run Time, T(n)	Complexity	Remarks
Case 1	$T(n) = c\frac{n(n-1)}{2}$	$T(n) = O(n^2)$	Best Case
Case 2	$T(n) = cn(n-1)$	$T(n) = O(n^2)$	Worst case
Case 3	$T(n) = \frac{3}{4}n(n-1)$	$T(n) = O(n^2)$	Average case

→ How do you make best case with (n-1) comparisons only ?

- By maintaining a variable flag, to check if there has been any swaps in a given pass.
- If not, the array is already sorted.

```
void bubble_sort(int x[], int n)
{
   int i,j;
   int flag=0;
   for(i=n-1; i>0; i--)
   {
     for(j=0; j<i; j++)
     {
       if(x[j]>x[j+1])
       {
         swap(&x[j], &x[j+1]);
         flag=1;
       }
     } if(flag==0) return;
   }
}
```

Efficient Sorting Algorithms

→ Two of the most popular sorting algorithms are based on divide-and-conquer approach.

- Quick sort
- Merge sort.

→ Basic concept of divide-and-conquer method:

```
Sort(list)
{
  if the list has length greater than ~~lowlist and~~ highlists 1.
  {
    Partition the list into lowlist and highlist;
    Sort(lowlist);
    Sort(highlist);
    combine(lowlist, highlist);
  }
}
```

Quick Sort - Algorithm

→ The algorithm was developed by a British computer scientist Tony Hoare in 1959.

→ The name "Quick Sort" comes from the fact that, quick sort is capable of sorting a list of data elements significantly faster (twice or thrice faster) than any of the common sorting algorithms.

→ It is one of the most efficient sorting algorithms and is based on the splitting of an array (partition) into smaller ones and swapping (exchange) based on the comparison with 'pivot' element selected.

→ Due to this, quick sort is also called as "Partition Exchange" sort. Like Merge sort, Quick sort also falls into the category of divide and conquer approach of problem-solving methodology.

→ Before diving into any algorithm, it's very much necessary for us to understand what are the real world applications of it. Quick sort provides a fast and methodical approach to sort any lists of things. Following are some of the applications where quick sort is used.

↳ Commercial computing: Used in various government and private organizations for the purpose of sorting various data like sorting of accounts/profiles by name or any given ID, sorting transactions by time or locations, sorting files by name or date of creation, etc.

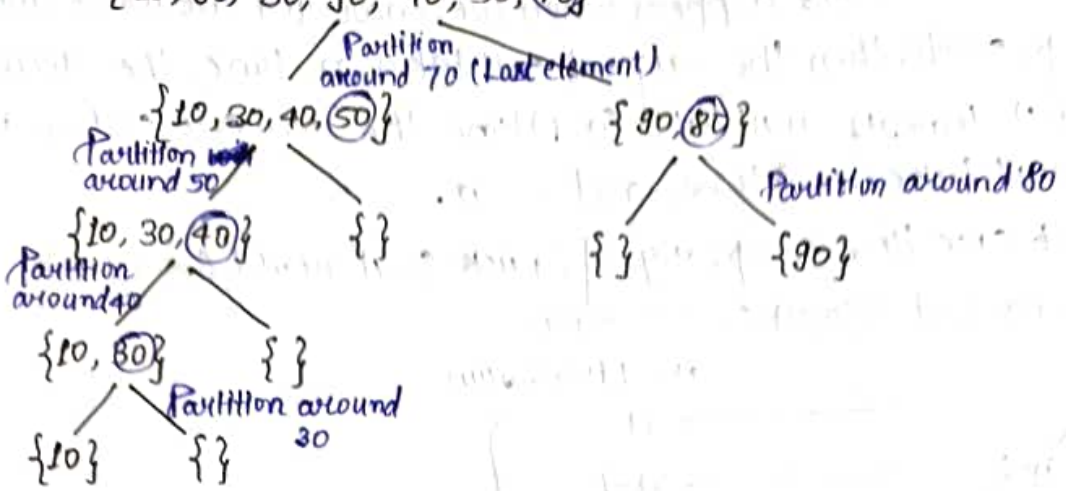
↳ Numerical computations: Most of the efficiently developed algorithms use priority queues and in turn sorting to achieve accuracy in all the calculations.

↳ Information Search: Sorting algorithms aid in better search of information and what faster way exists than to achieve sorting using quick sort.

↳ Basically, quick sort is used everywhere for faster results and in the cases where there are space constraints.

```
/* The main function that implements quick sort.
 * @ Parameters : array, starting index and ending index
 */

quickSort (arr[], low, high)
{
   if (low < high)
   {
     // pivot_index is partitioning index,
     // arr[pivot_index] is now at correct place in sorted array
     pivot_index = partition (arr, low, high);

     quickSort (arr, low, pivot_index -1);
     // Before pivot_index
     quickSort (arr, pivot_index +1, high);
     // After pivot_index
   }
  partition (arr[], low, high)
  {
     // pivot- Element at right most position
     pivot = arr[high];
     i= (low -1);  // Index of smaller element
     for (j= low; j<=high-1; j++)
     {
        // If current element is smaller than the pivot, swap the element
        // with pivot
        if (arr[j] < pivot)
        {
          i++;
          // Increment index of smaller element
          swap (arr[i], arr[j]);
        }
     }
     swap (arr[i+1], arr[high]);
     return (i+1);
  }
```

Eg. {10, 80, 30, 90, 40, 50, 70}



→ Best case scenario: When the partitions are as evenly balanced as possible, i.e, their sizes on either side of the pivot element are either equal or ~~are~~ have have size difference of 1 of each other.

→ Case-1: The case when sizes of sublist on either side of pivot becomes equal occurs when the subarray has an odd no. of elements and the pivot is right in the middle after partitioning. Each partition will have $\frac{(n-1)}{2}$ elements.

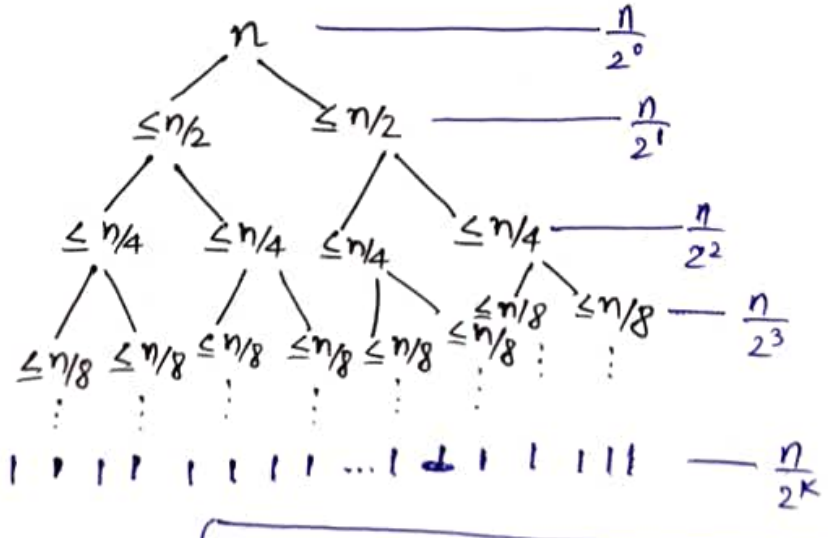
→ Case-2: The size difference of 1 between the two sublists on either side of pivot happens if the subarray has an even no., n, of elements. One partition will have $n/2$ elements with the other having $(n/2)-1$.

→ In either of these cases, each partition will have at most $(n/2)$ elements.

~~and the~~

Eg. Tree Representation of the subproblem sizes:

Quick sort: Best case scenario



After k steps, the value becomes 1.

$\Rightarrow \frac{n}{2^k} = 1$

$\Rightarrow n = 2^k$

Applying log on both sides,

$k = \log n$

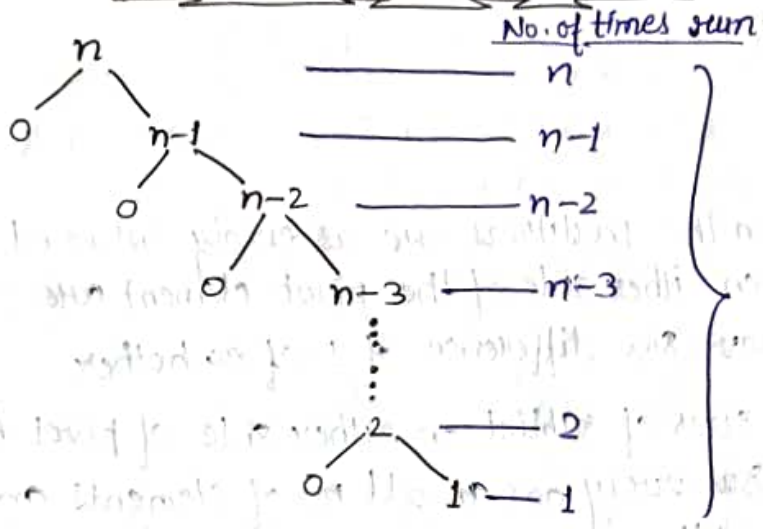
Time complexity = height of tree = $k = O(\log n)$

→ The best case complexity of quick sort algorithm is $O(\log n)$.

→ **Worst Case Scenario:** This happens when we encounter the most unbalanced partitions possible, then the original call takes n time, the recursive call on (n-1) elements will take (n-1) time, the recursive call on (n-2) elements will take (n-2) time, and so on.

The worst case time complexity of Quick Sort would be $O(n^2)$.

Quick Sort - Worst Case Scenario



Time complexity =
$n + (n-1) + (n-2) + (n-3) + \ldots + 2 + 1$
$= \sum n$
$= \frac{n(n-1)}{2}$
$\Rightarrow \boxed{O(n^2)}$.

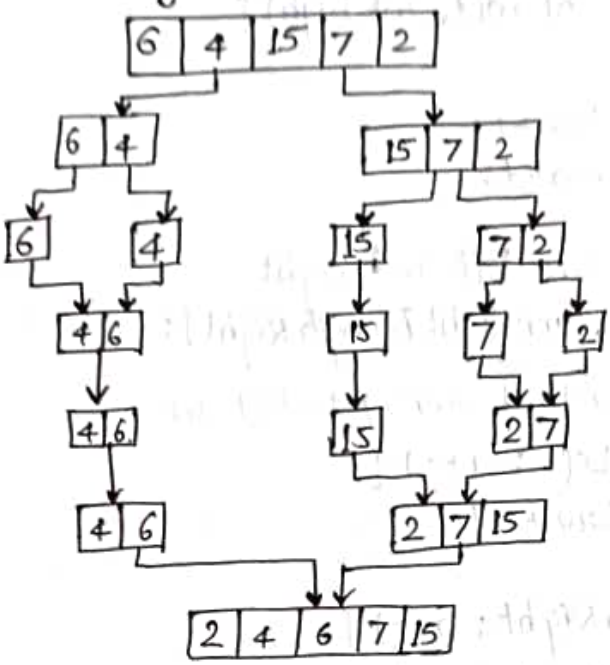
Space Complexity of Quick sort

→ The space complexity is calculated based on the space used in the recursive stack.

→ The worst case space used will be $O(n)$.
The average case space used will be of the order $O(\log n)$.

→ The worst case space complexity becomes $O(n)$, when the algorithm encounters its worst case where for getting a sorted list, we need to make n recursive calls.

Merge Sort Algorithm



Pass No.	Unsorted list	Divide/Merge	Sorted list
1	{6,4,15,7,2}	{6,4}, {15,7,2}	{}
2	{6,4}, {15,7,2}	{6}, {4}, {15}, {7,2}	{}
3	{6}, {4}, {15}, {7,2}	{4,6}, {15}, {7}, {2}	{4,6}
4	{4,6}, {15}, {7}, {2}	{4,6}, {15}, {2,7}	{4,6}, {15}, {27}
5	{4,6}, {15}, {2,7}	{4,6}, {2,7,15}	{4,6}, {2,7,15}
6	{4,6}, {2,7,15}	{2,4,6,7,15}	{2,4,6,7,15}
7	{}	{}	{2,4,6,7,15}
/	/	/	

```
#include <iostream>
using namespace std;

//Function to print the array
void printArray (int *arr, int length) {
    for (int i=0; i<length; i++) {
        cout <<arr[i] << " ";
    }
    cout <<endl;
}
```

```
//Function to merge the sub-arrays
void merge(int *arr, int low, int mid, int high){
    int i, j, k;
    int lengthLeft = mid - low+1;
    int lengthRight = high - mid;

//Creating two temp arrays to store left and right
    int arrLeft[lengthLeft], arrRight[lengthRight];

//Copying the data from the actual array to left an
    for(int a=0; a<lengthLeft; a++){
        arrLeft[a] = arr[low + a];
    }
    for(int a=0; a<lengthRight; a++){
        arrRight[a] = arr[mid+1+a];
    }

//Merging the temp arrays back into actual array
    i=0;  //starting index of arrLeft[]
    j=0;  //starting index of arrRight[]
    k=low;  //starting index of merged subarray
while(i<lengthLeft && j<lengthRight){
    //checking and placing the smaller number
    if(arrLeft[i] <= arrRight[j]){
        arr[k]= arrLeft[i];
        i++;
    } else{
        arr[k]= arrRight[j];
        j++;
    }
    k++;
}

//After the successful execution of the above loop
//copying the remaining elements of the left subarray
while(i<lengthLeft){
    arr[k] = arrLeft[i];
    k++;
    i++;
}
```

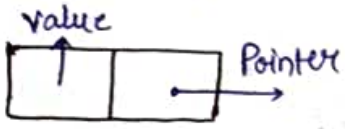
```
// copying the remaining elements of the right subarray
 while (j < lengthRight) {
        arr[k] = arrRight[j];
        k++;
        j++;
   }
}
// The mergeSort() function
void mergeSort (int * arr, int low, int high) {
     int mid;
      if (low < high) {
          // calculating the mid
          mid = (low + high) / 2;
// calling the function mergeSort() recursively and
          mergesort(arr, low, mid);
          mergeSort(arr, mid+1, high);
// calling the merge () function to merge the sorted array
          merge (arr, low, mid, high);
    }

int main () {
// Initializing the array
       int arr[] = {9, 14, 4, 6, 5, 8, 7};
// Calculating the length of the array
       int length = sizeof(arr) / sizeof(int);
 // Printing the array before Sorting
       cout << " Array, before calling the mergeSort(): ";
       printArray (arr, length);
 // calling the mergeSort() function
       mergeSort (arr, 0, length - 1);
  // Printing the array after sorting
       cout << " Array, after calling the mergeSort() : ";
       printArray (arr, length);

       return 0;
}
```

LINKED LISTS

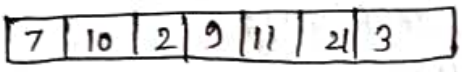
→ Arrays demand a contiguous memory location. Lengthening of an array is not possible.
- We would have to copy the whole array to some bigger memory location to lengthen its size.
- Similarly, inserting or deleting an element causes the elements to shift right and left, respectively.

→ But linked lists are stored in a non-contiguous memory location.
- To add a new element, we just have to create a node somewhere in the memory and get it pointed by the previous element.
- And for deleting an element, we just have to skip pointing to that particular node.

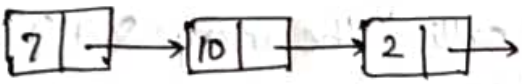
→ Structure: Every element in a linked list is called a node and consists of two parts, data and pointer.
- The data part stores the value, while the pointer part stores the pointer pointing to the address of the next node.



Node



→ In array, elements are stored in contiguous memory locations.



→ In linked lists, elements are stored in non-contiguous memory locations.

→ Drawbacks:
- Extra memory space for pointers is required (for every node, extra space for a pointer is needed).
- Random access is not allowed as elements are not stored in contiguous memory locations.

Implementations: Using structure

Head of the list → First entry

```
struct ListNode
{
    double value;
    ListNode *next;
};
int main(){
ListNode *head = NULL
head = new ListNode;  // allocate New Node
head → value = 12.5;  // store the value
head → next = NULL;   // Signify end of list

ListNode * secondPtr = new ListNode;
secondPtr→ value = 20;
head → next = second Ptr;

cout << " First item " << head →value << endl;
cout << " Second item " << secondPtr → value << endl;

return 0;
}
```

Constructor:

```
struct ListNode
{
    double value;
    ListNode *next;
    ListNode (double value1, ListNode *next1 = NULL)
    {
        value = value1;
        next = next1;
    }
};

ListNode *secondPtr = new ListNode (30);
ListNode * head = new ListNode (15, secondPtr);
```

Simpler version:

```
ListNode *head = new ListNode (30);
head = new ListNode (15, head);
```

Reading from file:

```
ListNode * numberList = NULL;
double number;
while (numberfile >> number)
{
  numberList = new ListNode (number, numberList);
}
```

(similar to cin >> number)

Roughly

```
ofstream file;
file = "new.txt";
```

Traversing a List

```
#include <iostream>
#include <fstream>
using namespace std;
struct ListNode
{
    double value;
    ListNode *next;

    ListNode (double value1, ListNode * next1=NULL)
    {
       value = value1;
       next = next1;
    }
};

ListNode *numberList =NULL;

int main()
{
    // open the file
    ifstream numberFile ("numberdata.txt");
    if (!numberFile)
    {
       cout << "Error in Opening the file";
       exit(1);
    }
```

numberdata.txt

```
30
40
50
60
```

```
    double number;
    cout<<" contents of the file are:";
    while (numberFile >> number)
     {
        cout << number << "  ";
        //create node to hold this number
        numberList = new ListNode (number, numberList);
     }
    ListNode *ptr = numberList;
    while (ptr != NULL)
    {
        cout << ptr->value << "  ";
        ptr = ptr->next;
     }
     return 0;
}
```

Output:

```
30    40    50   60
 60   50    40   30  → List.
```

Inserting Node

NumberList.h :

```
#include <iostream>
using namespace std;
class NumberList
 {
   protected:
     struct ListNode
     {
        double value;
        ListNode *next;
        ListNode (double value1, ListNode *next1 =NULL)
        { value =value1;
          next= next1;
        }
     };
     ListNode *head;
   public:
       Numberlist () { head = NULL; }
       ~NumberList ();
       void add (double number);
       void remove (double number);
       void displayList ();
 };
```

NumberList.cpp :

```
#include "NumberList.h"
using number namespace std;
void NumberList:: add (double number)
 {
    if (head == NULL)
       head = new ListNode (number);
    else
    {
      // List is not empty
      ListNode *nodePtr =head;
```

```
while (nodePtr->next!=NULL)
 {    nodePtr=nodePtr->next;
      //create a new node
      nodePtr->next = new ListNode(number);
  }
}

void NumberList::displayList()
 {
   ListNode *nodeptr=head;
   while (nodePtr)
    {
       //Print value in the current node
       cout<<nodePtr->value<<" ";
       //Move on to next node
       nodePtr=nodePtr->next;
    }
 }

//Destructor
NumberList::~NumberList()
{
    ListNode *nodePtr=head;  //start at head of the list
    while (nodePtr!=NULL)
    {
       //garbage keeps track of nodes to be deleted
       ListNode *garbage=nodePtr;
       nodePtr=nodePtr->next;
       //Delete the garbage node
       delete garbage;
    }
 }
```

Implementation:

```
#include "NumberList.h"
using namespace std;
int main()
{
    NumberList list;
    list.add(25);
    list.add(30);
    list.add(40);
    list.displayList();
    return 0;
}
```

Lists in a Sorted Order

SortedNumberList.h:

```
#include "NumberList.h"
class SortedNumberList : public NumberList
{
  public:
    void add(double number);
};

void SortedNumberList::add(double number)
{
    ListNode *nodePtr, *previousNodePtr;
    if(head==NULL || head->value >= number)
    {
      //The new node goes to the beginning of list
      head = new ListNode(number, head);
    }
    else
    {
      previousNodePtr = head;
      nodePtr = head->next;
      //Find the insertion point
      while(nodePtr!=NULL && nodePtr->value <number)
      {
```

```
        previousNodePtr = nodePtr;
        nodePtr = nodePtr->next;
      }
      //Insert the new node just before NodePtr
      previousNodePtr->next = new ListNode (number, nodePtr);
    }
  }
```

→ Removing a particular node from the list.

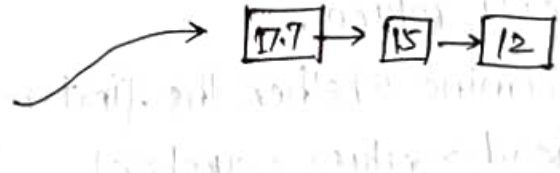
① Locating the node containing this number to be removed.
② Look into that node from the list.
③ Delete the memory allocated to that node.

```
void NumberList :: remove (double number)
{
    ListNode *previousNodePtr, *nodePtr;
    // If the list is empty, do nothing
    if (!head) return;
    // Determine whether the first node is to delete
    if (head->value == number)
    {
      nodePtr=head;
      head = head->next;
      delete nodePtr;
    }
    else
    {
    //Initialize nodePtr to head of the list
        nodePtr = head;

    //skip nodes value memeber is not given number
        while (nodePtr!=NULL && nodePtr->value != number)
        {
            previousNodePtr = nodePtr;
            nodePtr = nodePtr->next;
        }
```

```
// Link the previous node to the node after nodePtr
// then, delete nodePtr
 if (nodePtr)
  {
     previousNodePtr -> next = nodePtr -> next;
     delete nodePtr;
   } // if closed
 } // else is closed
} // function is closed
```

→ Remove() is common for both classes NumberList and SortedNumberList

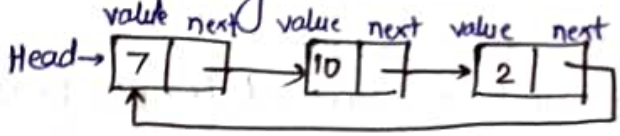
```
#include "NumberList.h"
using namespace std;
 int main()
 {
    NumberList list;
    //Build the list
    list.add (12);
    list.add (15);
    list.add (17.7);

    //Display the list
    cout << " The values of the list are \n";
     list.displayList();
     cout << "Removing the node";
     list.remove(15);
     list.displayList();
     return 0;
  }
```



Circular Linked List

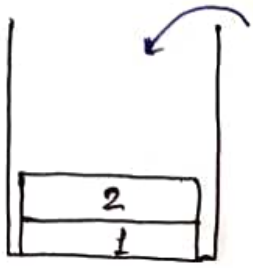
↳ A linked list where the last element points to the first element (head), hence forming a circular chain.

↳ There is no node pointing to the NULL, indicating the absence of any end node.



STACKS

- A linear data structure
- Any operation on the stack is performed in LIFO (Last In First Out) order, ie., the element to enter the container last would be the first one to leave the container.
- Elements above an element must be removed first before fetching any element.



Insertion/deletion operation can happen on this end

Item 2 which entered the basket last will be the first one to come out.

Implementation using an array:

Instack.h:

```
class Instack
{
  private:
    int * stackArray;
    int capacity;
    int top;
  public:
     Instack (int cpty);
     ~Instack () { delete [] stackArray; }
     void push (int value);
     void pop (int &value);
     bool isEmpty ();
};
```

→ `int stackArray[50];` ✗
Dynamic memory allocation

Eg:
```
int *new = new int[5];
delete [] new;
```

Instack.cpp :

```
#include "Instack.h"
Instack::Instack (int cpty)
{
   this->capacity = cpty;
   stackArray = new int [capacity];
   top= 0;
}
Instack:: push (int value)
{
   if (top== capacity) { cerr<<" stack is Full "; exit(1); }
   stackArray [top]= value;
   top++;
}
bool Instack:: isEmpty()
{
    if (top= =0)
       return true;
     else
         return false;
}
void Instack:: pop(int &value)
{
    if (isEmpty()) { cerr <<"Stack is Empty"; exit(1); }
    top--;
    value=stackArray [top];
}
```

main.cpp :

```
#include "Instack.h"
#include <iostream>
Using namespace std;
```

```
int main()
{
    Instack stack(5);
    int value [] = {5, 15, 25, 35, 45};
    int value;
    cout << " Pushing \n";
    for (int k=0; k<5; ++k)
    {
       cout << value [k] << "  ";
       stack.push(value [k]);
    }
    cout << " Pop out " << endl;
    while (! stack.isEmpty())
    {
       stack. pop(value);
       cout << value;
    }
    cout << endl;
    return 0;
}
```

Implementation using Linked List:

DynIntStack.h:

```
#ifndef DYNINTSTACK_H
#define DYNINTSTACK_H

class DynIntstack
{
    private:
        class stackNode
        {
          friend class DynIntstack;
          int value;
          stackNode *next;
          // Constructor
          stackNode (int value1, stackNode * next1 = NULL)
          {
              value = value1;
              next = next1;
          }
        };
        stackNode * top;
    public:
        DynIntStack() { top= NULL; }
        void push(int);
        void pop(int &);
        bool isEmpty();
};
#endif
```

DynIntStack.cpp:

```
#include "DynIntstack.h"
#include <iostream>
using namespace std;
void DynIntStack :: push (int num)
{
    top= new stackNode (num, top);
}
```

```
void DynIntStack::pop(int &num)
{
   StackNode *temp;
   if (isEmpty())
   {
      cout << "Stack is Empty";
      exit(1);
   else
   {
      num = top->value;
      temp = top;
      top = top->next;
      delete temp;
   }
}

bool DynIntStack:: isEmpty()
{
    if(!top)
       return true;
    else
       return false;
}
```

main.cpp :

```
#include "DynIntStack.h"
#include <iostream>
using namespace std;
int main()
{
   DynIntStack stack;
   int var;
   stack.push(45);
   Stack.push(30);
   stack.pop(var);
   Stack.pop(var);
   return 0;
}
```

STL Stack Container

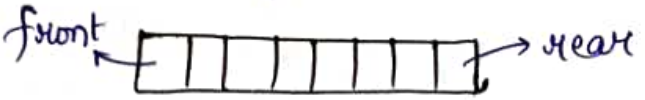
↳ Standard Template Library

Eg:

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
int main()
{
    stack <int, vector<int>> istack;
    for (int x=2; x<8; x+=2)
    {
        istack.push (x);
    }
    cout<< istack.size () <<endl;
    while (! istack.empty() )
        istack.pop();
    return 0;
}
```

QUEUE ADT

- FIFO (First In First Out)
- We have to maintain both the ends because we have insertion at one end and deletion from the other end.

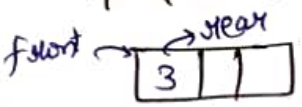


Methods:

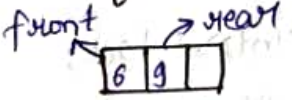
1. Enqueue
2. Dequeue

Eg:

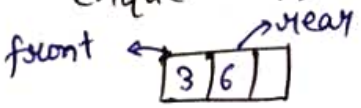
enqueue(3);



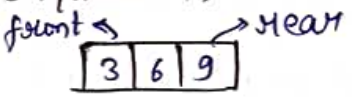
dequeue();



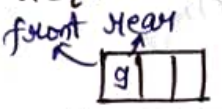
enqueue(6);



enqueue(9);

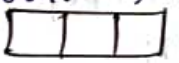


dequeue();



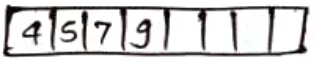
dequeue();

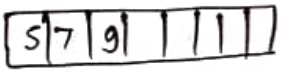
front = -1, rear = -1



Eg:

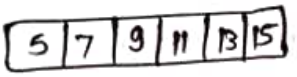
Deque

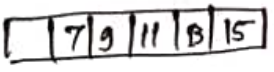


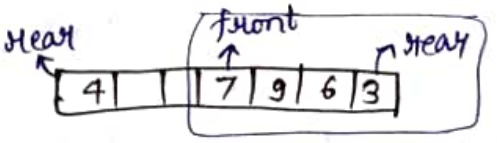


Eg:

Deque








```
if (rear == queueSize - 1)
    rear = 0;
else
    rear ++
```

rear = (rear + 1) % queueSize;

Queue is Full ⇒ (rear + 1) % queueSize == front;

Array Implementation of Queue: Static Queue

Methods:

① Insertion (enqueue):
- ~~Insert~~ Increment rear by 1.
- Insert the element
- Time complexity: O(1).

② Deletion (dequeue):
- Remove the element at zeroth index
- Shift all other elements to their immediate left.
- Decrement rear by 1.

OR

- Remove the element at the zeroth index (no need for that in an array
- Increment front by 1.
- Time complexity: O(1).

- our first element is at index front +1, and the nearest element is at rear.
- Condition for queue empty: front = rear.
- Condition for queue full: ~~rear~~ rear = size-1.

IntQueue.h :

```
#ifndef INTQUEUE_H
#define INTQUEUE_H

class IntQueue
{
  private:
      int *queueArray;
      int queueSize;
      int front;
      int rear;
      int numItems;
  public:
      IntQueue(int);
      ~IntQueue();
      void enqueue(int);
      void dequeue(int &);
```

```
    bool isEmpty();
    bool isFull();
    void clear();
};
#endif
```

IntQueue.cpp:

```
IntQueue::IntQueue(int s)
{
    queueArray = new int[s];
    queueSize = s;
    front = -1;
    rear = -1;
    numItems = 0;
}
IntQueue::~IntQueue()
{
  delete[] queueArray;
}
void IntQueue::enqueue(int num)
{
    if (isFull())
    {
      cout << "Queue is Full " << endl;
      exit(1);
    }
    else
    {
      rear = (rear+1) % queueSize;
      queueArray[rear] = num;
      numItems++;
    }
    void IntQueue::dequeue(int &num)
    {
      if (isEmpty())
      {
        cout << "Dequeue cannot be done as queue is Empty " << endl;
        exit(1);
      }
```

```
    else
    {
        front = (front + 1) % queueSize;
        num = queueArray[front];
        numItems--;
    }
}

bool IntQueue::isEmpty()
{
    if (numItems > 0)
        return false;
    else
        return true;
}

void IntQueue::clear()
{
    front = -1;
    rear = -1;
    numItems = 0;
}
```

main.cpp :

```
#include "IntQueue.h"
#include <iostream>
using namespace std;
int main()
{
    IntQueue iQueue(5);
    int value;
    for (int K=0; K<5; ++K)
        iQueue.enqueue(K*K);
    while (!iQueue.isEmpty())
    {
      iQueue.dequeue(value);
      cout << value <<" ";
    }
    return 0;
}
```

Linked list Implementation of Queue : Dynamic Queue

① Dynamic queue starts with empty linked list.
② At first enqueue, a node is added, which is pointed to by front and rear pointers.
③ A new node is added to rear of the list, rear pointer is updated to the next node.

DynIntQueue.h

```
#ifndef DYNINTQUEUE_H
#define DYNINTQUEUE_H
 class DynIntQueue
 {
    private:
       class QueueNode
       {
          friend class DynIntQueue;
          int value;
          QueueNode *next;
          QueueNode (int value1, QueueNode *next1=NULL)
          {
             value = value1;
             next = next1;
          }
       };
       QueueNode *front;
       QueueNode *rear;
    public:
       DynIntQueue();
       ~DynIntQueue();
        void enqueue(int);
        void dequeue (int &);
        bool isEmpty();
        void clear();
 };
#endif
```

DynIntQueue.cpp:

```
#include "DynIntQueue.h"
#include <iostream>
using namespace std;

DynIntQueue::DynIntQueue()
{
  front = NULL;
  rear = NULL;
}
DynIntQueue::~DynIntQueue()
{
  clear();
}
void DynIntQueue::enqueue(int num)
{
    if(isEmpty())
    {
      front = new QueueNode(num);
      rear = front;
    }
    else
    {
      rear->next = new QueueNode(num);
      rear = rear->next;
    }
}
void DynIntQueue::dequeue(int &num)
{
   QueueNode *temp; //create temporary pointer variable
   if(isEmpty())
   {
     cout << " The queue is empty \n";
     exit(1);
   }
   else
    {
       num= front->value;
```

```
        temp = front;
        front = front -> next;
        delete temp;
    }
}
bool DynIntQueue:: isEmpty()
{
    if (front == NULL)
        return true;
    else
        return false;
    }
void DynIntQueue:: clear()
{
    int value;
    while (! isEmpty())
    {
        dequeue (value);
    }
}
```

main.cpp :

```
#include "DynIntStack.h"
#include <iostream>
using namespace std;
int main ()
{
    DynIntQueue iQueue;
    cout << " Enqueuing the items \n";
    for (int K=1; k <=5; k++)
        iQueue . enqueue (k*k);
    cout << " The values in the Queue are \n";
    while (!iQueue . isEmpty())
    {
        int value;
        iQueue . dequeue (value);
```

```
        cout << value << "  ";
    }
    return 0;
}
```