Sorting

- Bubble sort
- Insertion sort
- Selection Sort
- Quick sort
- Merge sort

Bubble sort

- simplest sorting algorithm
- repeatedly iterate through the array and swap adjacent elements that are unordered.
- repeat this until the array is sorted.

Why bubble sort is called "bubble" sort?

because the elements with greater value than the adjacent element "bubble" towards the end.

Example---1

5	4	3	2	1
4	5	3	2	1
4	3	5	2	1
4	3	2	5	1
4	3	2	1	5

4	3	2	1
3	4	2	1
3	2	4	1
3	2	1	4

3	2	1
2	3	1
2	1	3

2	1
1	2

Psuedo code

```
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)

        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(arr[j], arr[j+1]);
}
```

Example---1-----iteration----1

0	1	2	3	4	I	J
5	4	3	2	1	0	0
4	5	3	2	1	0	1
4	3	5	2	1	0	2
4	3	2	5	1	0	3
4	3	2	1	5		

N=5
i<n-1

j<n-1-I
i==0
J varies between 0 to 3

```
(arr[j] >
arr[j+1])
```

Example---1-----iteration----2

0	1	2	3	4	I	J
5	4	3	2	1	0	0
4	5	3	2	1	0	1
4	3	5	2	1	0	2
4	3	2	5	1	0	3
4	3	2	1	5		
4	3	2	1	5	1	0
3	4	2	1	5	1	1
3	2	4	1	5	1	2
3	2	1	4	5		

N=5
i<n-1

j<n-1-I
i==1
J varies between 0 to 2

```
(arr[j] >
arr[j+1])
```

Example---1-----iteration----3

0	1	2	3	4	I	J
5	4	3	2	1	0	0
4	5	3	2	1	0	1
4	3	5	2	1	0	2
4	3	2	5	1	0	3
4	3	2	1	5		
4	3	2	1	5	1	0
3	4	2	1	5	1	1
3	2	4	1	5	1	2
3	2	1	4	5		
3	2	1	4	5	2	0
2	3	1	4	5	2	1
2	1	3	4	5		

N=5
i<n-1

j<n-1-I
i==2
J varies between 0 to 1

```
(arr[j] >
arr[j+1])
```

Example---1-----iteration----4

0	1	2	3	4	I	J
5	4	3	2	1	0	0
4	5	3	2	1	0	1
4	3	5	2	1	0	2
4	3	2	5	1	0	3
4	3	2	1	5		
4	3	2	1	5	1	0
3	4	2	1	5	1	1
3	2	4	1	5	1	2
3	2	1	4	5		
3	2	1	4	5	2	0
2	3	1	4	5	2	1
2	1	3	4	5		
2	1	3	4	5	3	0
1	2	3	4	5	3	

N=5
i<n-1

j<n-1-I
i==3
J varies between 0 to 1

```
(arr[j] >
arr[j+1])
```

Characteristics of Bubble Sort:

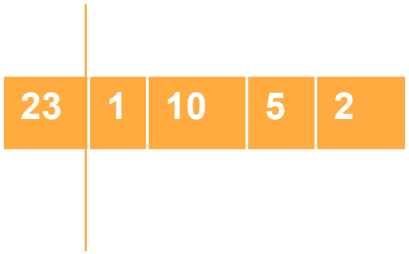
- Large values are always sorted first.
- It only takes one iteration to detect that a collection is already sorted.
- The best time complexity for Bubble Sort is O(n). The average and worst time complexity is $O(n^2)$.
- The space complexity for Bubble Sort is O(1), because only single additional memory space is required.

Insertion sort

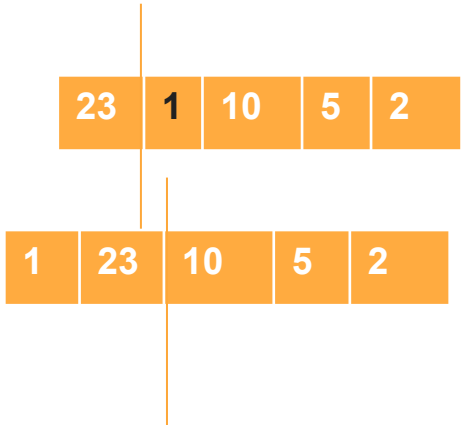
- Insertion sort is a simple sorting algorithm **that is appropriate** for small inputs.
  - Most common sorting technique used by card players.
- The list is divided into two parts: sorted and unsorted.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.

Example

- Initial

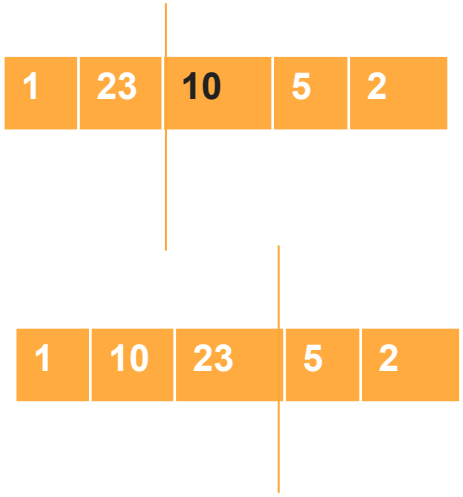


- First iteration

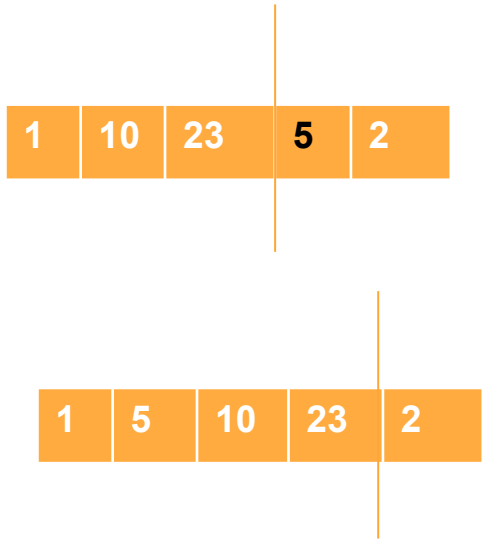


Example

- second iteration

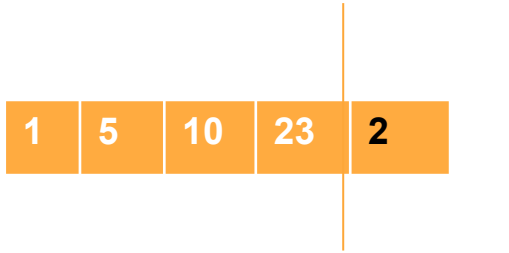


- Third iteration



Example

- Fourth iteration



Algorithm

```
void insertionSort(int arr[], int n)
{
   for (j = 1; j < n; i++)
   {
        key = arr[j];
        i = j - 1;
      //insert a[j] into the sorted
sequence
        while (i>=0) && arr[i] > key)
do
      {   arr[i + 1] = arr[i];
                 i=i-1;
               }
        arr[i + 1] = key;
    }
}
```

- N=5;;J<n

0	1	2	3	4	j	key	l
23	1	10	5	2	1	1	0
23	23	10	5	2	1	1	-1
1	23	10	5	2	1	1	-1

```
void insertionSort(int arr[], int n)
{
  for (j = 1; j < n; i++)
  {
      key = arr[j];
      i = j - 1;
    //insert a[j] into the sorted
sequence
      while (i>=0) && arr[i] > key)
do
     {   arr[i + 1] = arr[i];
                i=i-1;
              }
      arr[i + 1] = key;
   }
}
```

Example

- N=5::J<n

0	1	2	3	4	j	key	l
23	1	10	5	2	1	1	0
23	23	10	5	2	1	1	-1
1	23	10	5	2	1	1	-1
1	23	23	5	2	2	10	1
1	10	23	5	2	2	10	0
1	10	23	23	2	3	5	2
1	10	10	23	2	3	5	1
1	5	10	23	2	3	5	0
1	5	10	23	23	4	2	3
1	5	10	10	23	4	2	2
1	5	5	10	23	4	2	1
1	2	5	10	23	4	2	0

```
void insertionSort(int arr[], int n)
{
  for (j = 1; j < n; j++)
  {
      key = arr[j];
      i = j - 1;
    //insert a[j] into the sorted
sequence
      while (i>=0) && arr[i] > key)
do
     {   arr[i + 1] = arr[i];
                i=i-1;
              }
      arr[i + 1] = key;
   }
}
```

c1	for (j = 1; j < n; j++)	n
c2	key = arr[j];	n-1
c3	i = j - 1;	n-1
c4	//insert a[j] into the sorte sequence	n-1
c5	while (i>=0) && arr[i] > key)	$(i=2 \text{ to } n)\sum t_j$
c6	arr[i + 1] = arr[i];	$(i=2 \text{ to } n)\sum t_j-1$
c7	i=i-1;	$(i=2 \text{ to } n)\sum t_j-1$
c8	arr[i + 1] = key;	n-1

Characteristics of Insertion Sort:

- It is efficient for smaller data sets, but very inefficient for the larger lists.
- Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
- Its space complexity is less. Insertion sort requires a single additional memory space.
- Overall time complexity of Insertion sort is $O(n^2)$.

Insertion sort visualization

https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/visualize/

Selection sort

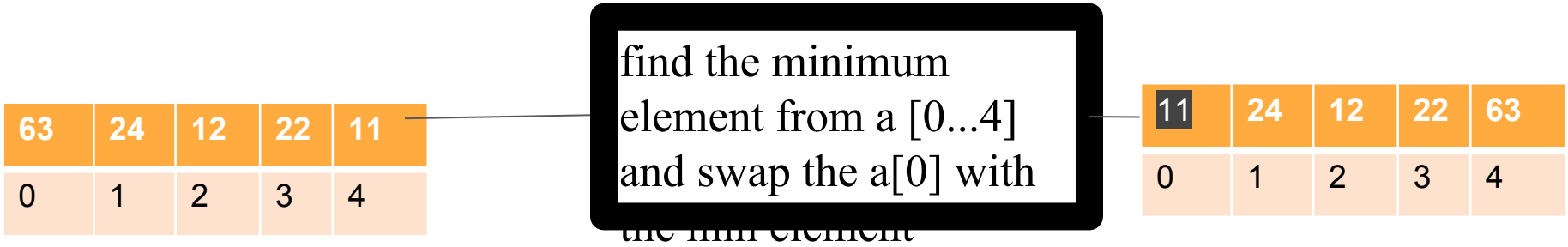
The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

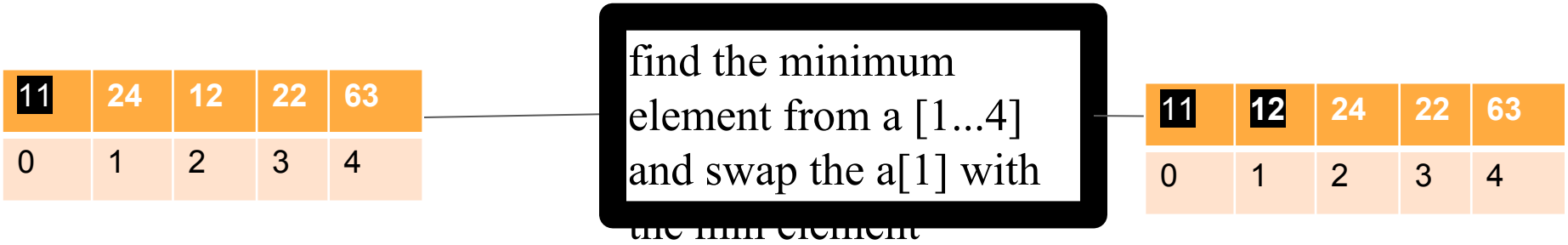
1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

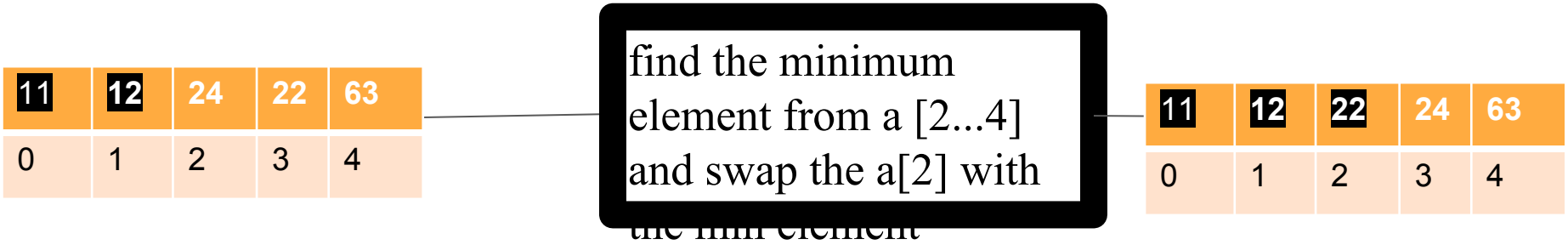
Example


63 24 12 22 11
0 1 2 3 4
find the minimum element from a [0...4] and swap the a[0] with
11 24 12 22 63
0 1 2 3 4

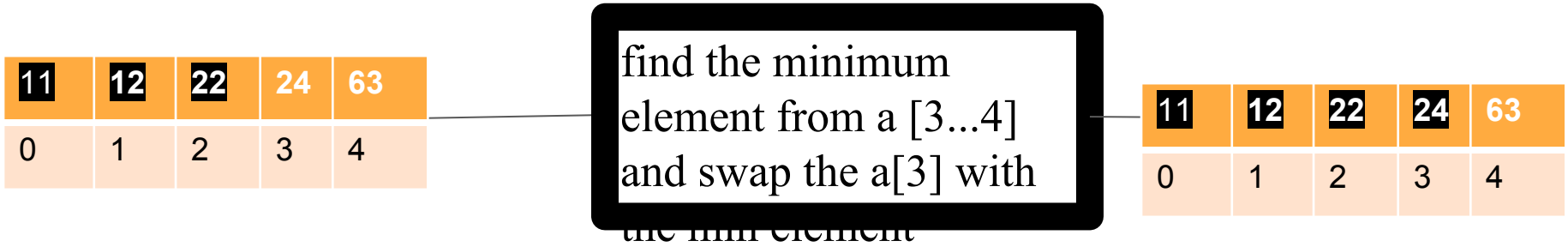
Example


11 24 12 22 63
0 1 2 3 4
find the minimum element from a [1...4] and swap the a[1] with
11 12 24 22 63
0 1 2 3 4

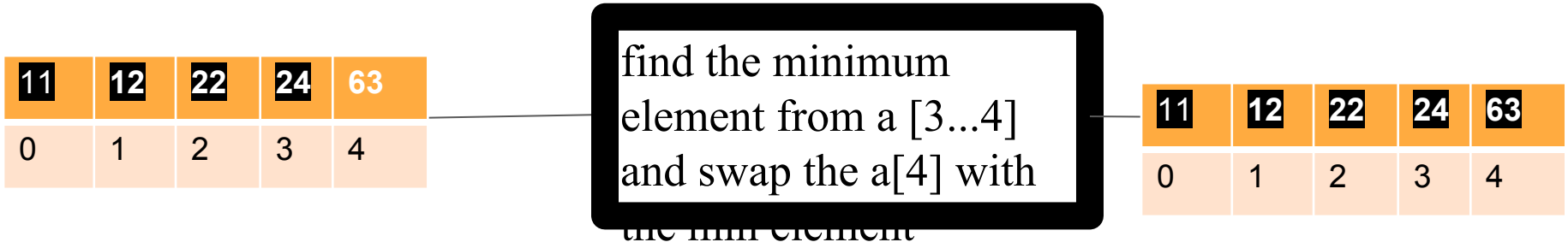
Example


11 12 24 22 63
0 1 2 3 4
find the minimum element from a [2...4] and swap the a[2] with
11 12 22 24 63
0 1 2 3 4

Example


11 12 22 24 63
0 1 2 3 4
find the minimum element from a [3...4] and swap the a[3] with
11 12 22 24 63
0 1 2 3 4

Example


11 12 22 24 63
0 1 2 3 4
find the minimum element from a [3...4] and swap the a[4] with
11 12 22 24 63
0 1 2 3 4

Pseudo code

```
void selectionSort(int arr[], int n)
{
    int i, j, minidx;
      // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        minidx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[minidx])
            minidx = j;
          // Swap the found minimum element with the first element
        swap(arr[minidx], arr[i]);
    }
}
```

I iteration

- N=5;; i<n-1

0	1	2	3	4	i	minidx	j
63	24	12	22	11	0	0	1
					0	1	2
					0	2	3
						4	4
11	24	12	22	63		4	

```
void selectionSort(int arr[], int n)
{
    int i, j, minidx;
      // One by one move boundary of
unsorted subarray
    for (i = 0; i < n-1; i++)
   {
        // Find the minimum element in
unsorted array
        minidx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[minidx])
            minidx = j;
 // Swap the found minimum element with
the first element
        swap(arr[minidx], arr[i]);
    }
}
```

II iteration

0	1	2	3	4	i	minidx	j
63	24	12	22	11	0	0	1
					0	1	2
					0	2	3
						4	4
11	24	12	22	63		4	
11	24	12	22	63	1	1	2
						2	3
							4
11	12	24	22	63			

```
void selectionSort(int arr[], int n)
{
    int i, j, minidx;
      // One by one move boundary of
unsorted subarray
    for (i = 0; i < n-1; i++)
   {
        // Find the minimum element in
unsorted array
        minidx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[minidx])
            minidx = j;
 // Swap the found minimum element with
the first element
        swap(arr[minidx], arr[i]);
    }
}
```

III iteration

0	1	2	3	4	i	min idx	j
63	24	12	22	11	0	0	1
					0	1	2
					0	2	3
						4	4
11	24	12	22	63		4	
11	24	12	22	63	1	1	2
						2	3
							4
11	12	24	22	63	2	2	3
						3	4
11	12	22	24	63			

```
void selectionSort(int arr[], int n)
{
    int i, j, minidx;
      // One by one move boundary of
unsorted subarray
    for (i = 0; i < n-1; i++)
   {
        // Find the minimum element in
unsorted array
        minidx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[minidx])
            minidx = j;
 // Swap the found minimum element with
the first element
        swap(arr[minidx], arr[i]);
    }
}
```

IV iteration

0	1	2	3	4	i	minidx	j
63	24	12	22	11	0	0	1
					0	1	2
					0	2	3
						4	4
11	24	12	22	63		4	
11	24	12	22	63	1	1	2
						2	3
							4
11	12	24	22	63	2	2	3
						3	4
11	12	22	24	63	3	4	4
					4		

```
void selectionSort(int arr[], int n)
{
    int i, j, minidx;
      // One by one move boundary of
unsorted subarray
    for (i = 0; i < n-1; i++)
   {
        // Find the minimum element in
unsorted array
        minidx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[minidx])
            minidx = j;
 // Swap the found minimum element with
the first element
        swap(arr[minidx], arr[i]);
    }
}
```

Characteristics of Selection Sort:

- It is efficient for smaller data sets, but very inefficient for the larger lists.
- Selection Sort is efficient in average case which reduces the number of comparisons.
- Its space complexity is less. selection sort requires a single additional memory space.
- Overall time complexity of selection sort is $O(n^2)$.

Visualization effect

https://www.hackerearth.com/practice/algorithms/sorting/insertion-sort/visualize/

Quick sort

- Quick sort is one of the most efficient sorting algorithms.
- works on the principle of **Divide and Conquer**.
  - divide the problem into sub-problems
  - conquer the sub-problems

- Efficient than Merge sort by using partitioning technique

Partition technique

Example

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
65	70	75	80	85	60	55	50	45
65	45	75	80	85	60	55	50	70
65	45	50	80	85	60	55	75	70
65	45	50	55	85	60	80	75	70
65	45	50	55	60	85	80	75	70
60	45	50	55	65	85	80	75	70

Pseudo code

```
void quicksort(int a[25],int first, int last)
{

   if(first<last){
      pivot=first;
      i=first;
      j=last;

      while(i<j)
      {
         while(a[i]<=a[pivot]&&i<last)
            i++;
         while(a[j]>a[pivot])
            j--;
            if(i<j)
         {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
         }
      }

      temp=a[pivot];
      a[pivot]=a[j];
      a[j]=temp;
      quicksort(a,first,j-1);
      quicksort(a,j+1,last);

   }
}
```

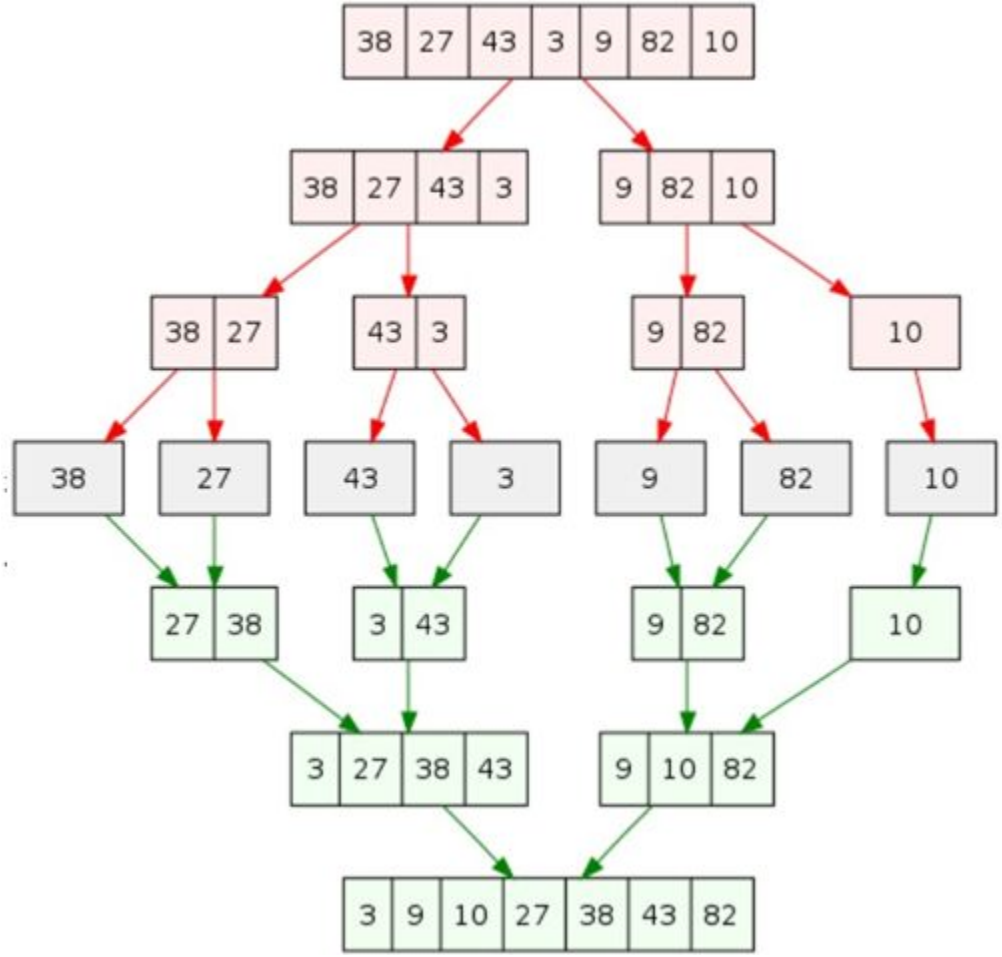
Characteristics of Quick Sort

- Quick Sort is useful for sorting arrays of any size.
- Time complexity of Quick Sort is O(nLogn).
- Space complexity of Quick Sort is O(nLogn).

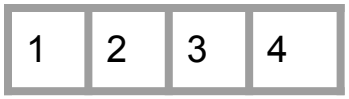
MERGE sort

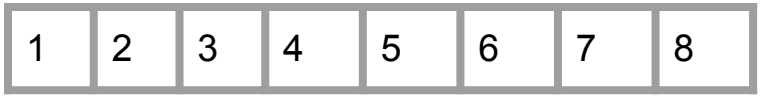
- Merge sort is one of the most efficient sorting algorithms.
- works on the principle of Divide and Conquer.
- Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Example


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

merge the sorted array

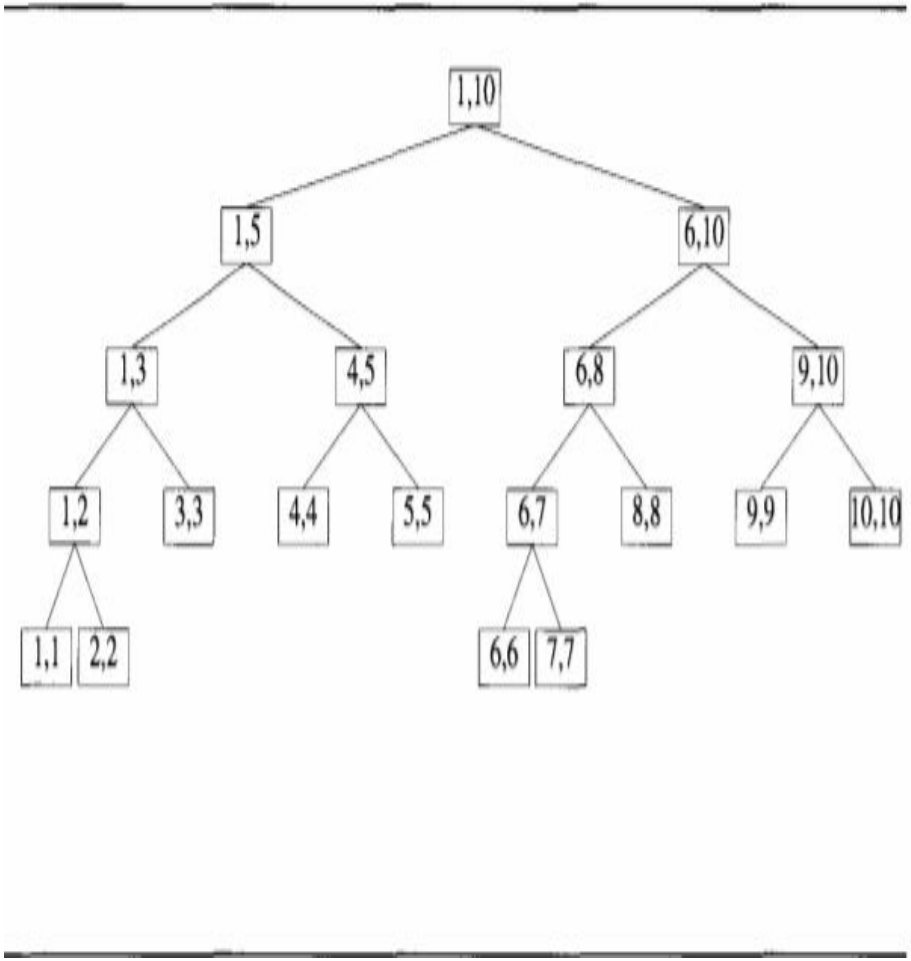


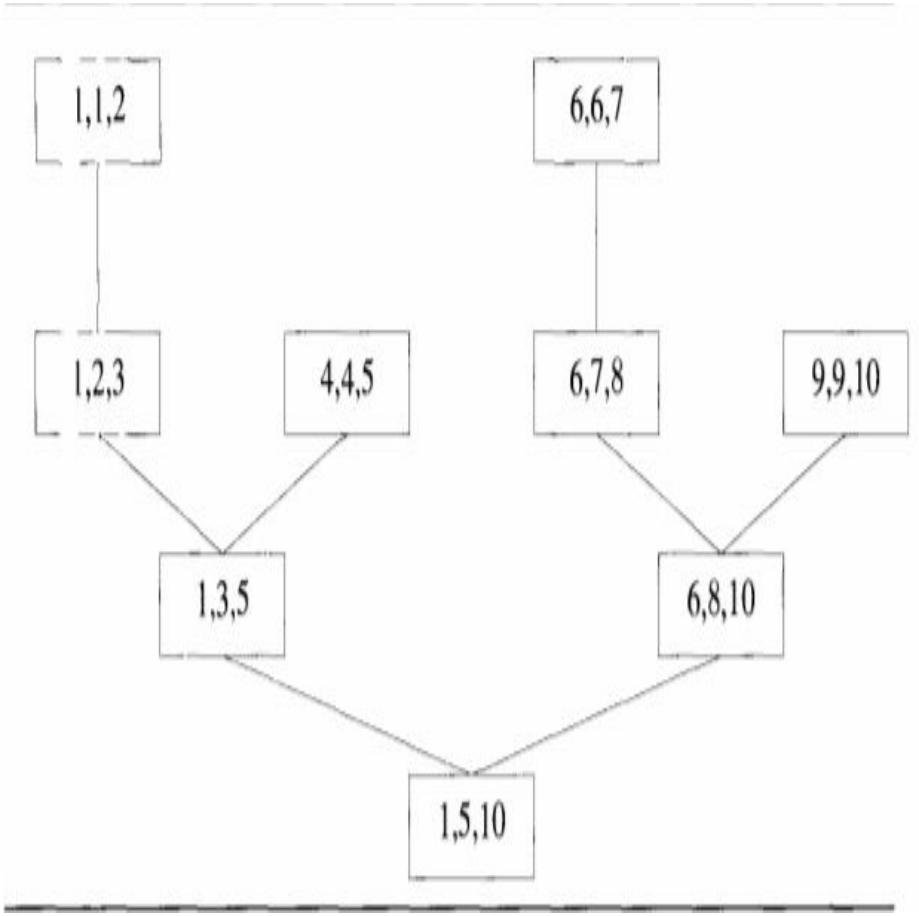


Psuedo code

```
1   Algorithm MergeSort(low, high)
2   // a[low : high] is a global array to be sorted.
3   // Small(P) is true if there is only one element
4   // to sort. In this case the list is already sorted.
5   {
6       if (low < high) then  // If there are more than one element
7       {
8           // Divide P into subproblems.
9               // Find where to split the set.
10                  mid := ⌊(low + high)/2⌋;
11          // Solve the subproblems.
12              MergeSort(low, mid);
13              MergeSort(mid + 1, high);
14          // Combine the solutions.
15              Merge(low, mid, high);
16      }
17  }
```

tree calls for mergesort(1,10)



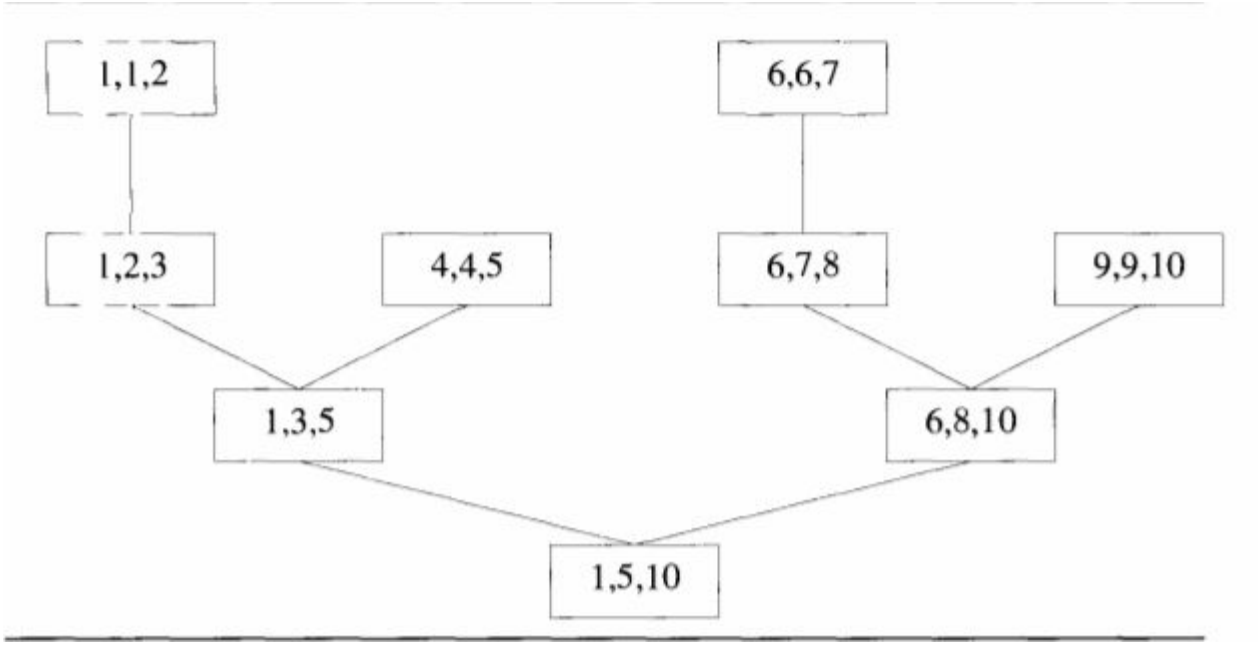


Psue

```
1   Algorithm Merge(low, mid, high)
2   // a[low : high] is a global array containing two sorted
3   // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4   // is to merge these two sets into a single set residing
5   // in a[low : high]. b[ ] is an auxiliary global array.
6   {
7       h := low; i := low; j := mid + 1;
8       while ((h ≤ mid) and (j ≤ high)) do
9       {
10          if (a[h] ≤ a[j]) then
11          {
12              b[i] := a[h];h := h + 1;
13          }
14          else
15          {
16              b[i] := a[j]; j := j + 1;
17          }
18          i := i + 1;
19      }
20      if (h > mid) then
21          for k := j to high do
22          {
23              b[i] := a[k]; i := i + 1;
24          }
25      else
26          for k := h to mid do
27          {
28              b[i] := a[k]; i := i + 1;
29          }
30      for k := low to high do a[k] := b[k];
31  }
```

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

Tree calls for Merge algorithm



Characteristics of merge sort

- efficient
- handle large volume of data
- worst case analysis O(nlogn)
- space complexity is high due to the usage of auxiliary array of same size

visualization

https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/visualize/

Hashing

- What is Hashing?
- What is Hash Function? (k mod 10, k mod n, mid square method, etc)
- What is Hash Table?
  - array , linked list, associative memory

key---24,52,91,67,48,83

Hashing

- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- It is a technique to convert a range of key values into a range of indexes of an array.
- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

- In a hashed search, the key, through an algorithmic function, determines the location of the data. Because we are searching an array, we use a hashing algorithm to transform the key into the index that contains the data we need to locate.

- Another way to describe hashing is as a key-to-address transformation in which the keys map to addresses in a list

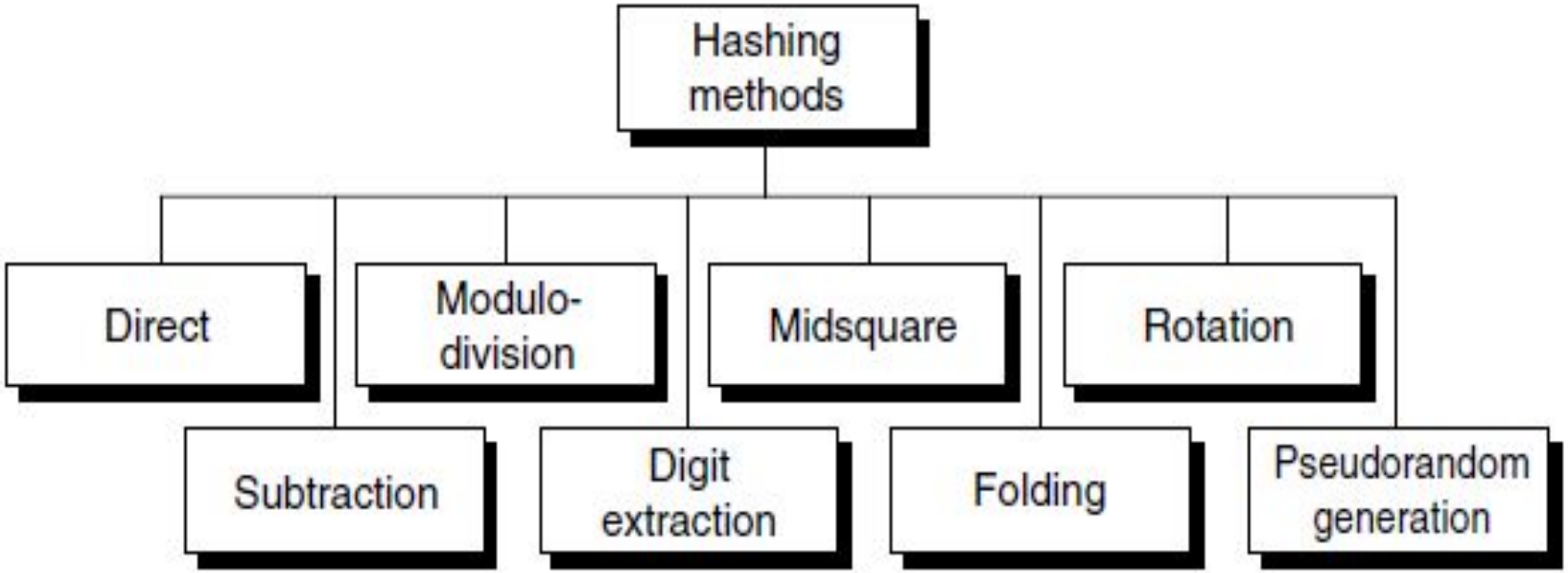
- Hashing is a key-to-address mapping process.

Hash Function

- A fixed process converts a key to a hash key is known as a **Hash Function.**
- This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash.**
- Hash value represents the original string of characters, but it is normally smaller than the original.

Hash Table

- Hash table or hash map is a data structure used to store key-value pairs.
- Each key is mapped to a value in the hash table. The keys are used for indexing the values/data
- It is a collection of items stored to make it easy to find them later.
- It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
- It is an array of list where each list is known as bucket.
- It contains value based on the key.



Basic Hashing Techniques

Direct addressing

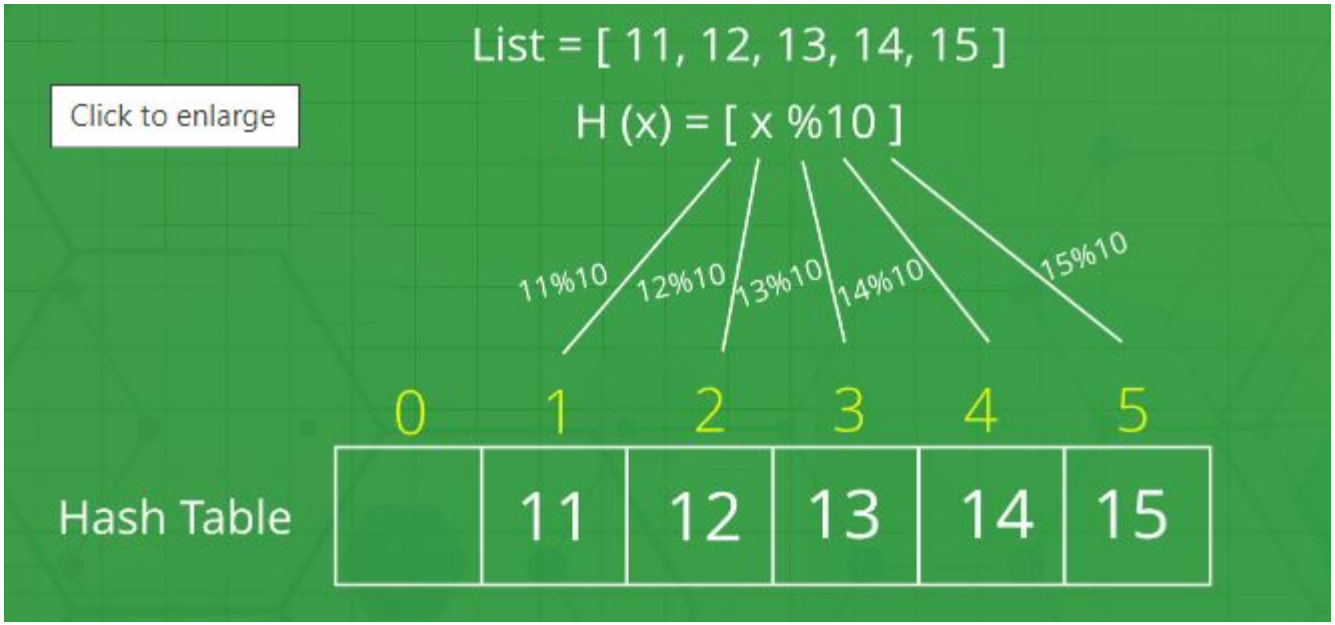
Direct addressing is possible only when we can afford to allocate an array that has one position for every possible key.

every key has a unique array position, searching takes a constant time. O(1)

Modulo-division Hashing

- address = key MODULO listSize

Modulo-division Hashing---example-1



Example--2

set of strings {“ab”, “cd”, “efg”}

- table of size 7
- *Assign numerical value to every character----here positional value is assigned*
- *“ab” = 1 + 2 = 3,*
- *“cd” = 3 + 4 = 7 ,*
- *“efg” = 5 + 6 + 7 = 18*

- “ab” in 3 mod 7 = 3,
- “cd” in 7 mod 7 = 0, and
- “efg” in 18 mod 7 = 4.

0	1	2	3	4	5	6
cd			ab	efg		

Digit-Extraction Method

Selected digits are extracted from the key and used as the address

379452 ⇨ 394
121267 ⇨ 112
378845 ⇨ 388
160252 ⇨ 102
045128 ⇨ 051

Midsquare Method

The key is squared and the address is selected from the middle of the squared number

$9452^2 = 89340304$: address is 3403

Variation on the midsquare

Select a portion of the key and then use them rather than the whole key.

Doing so allows the method to be used when the key is too large to square

379452: 379^2 = 143641 ⇨ 364
121267: 121^2 = 014641 ⇨ 464
378845: 378^2 = 142884 ⇨ 288
160252: 160^2 = 025600 ⇨ 560
045128: 045^2 = 002025 ⇨ 202

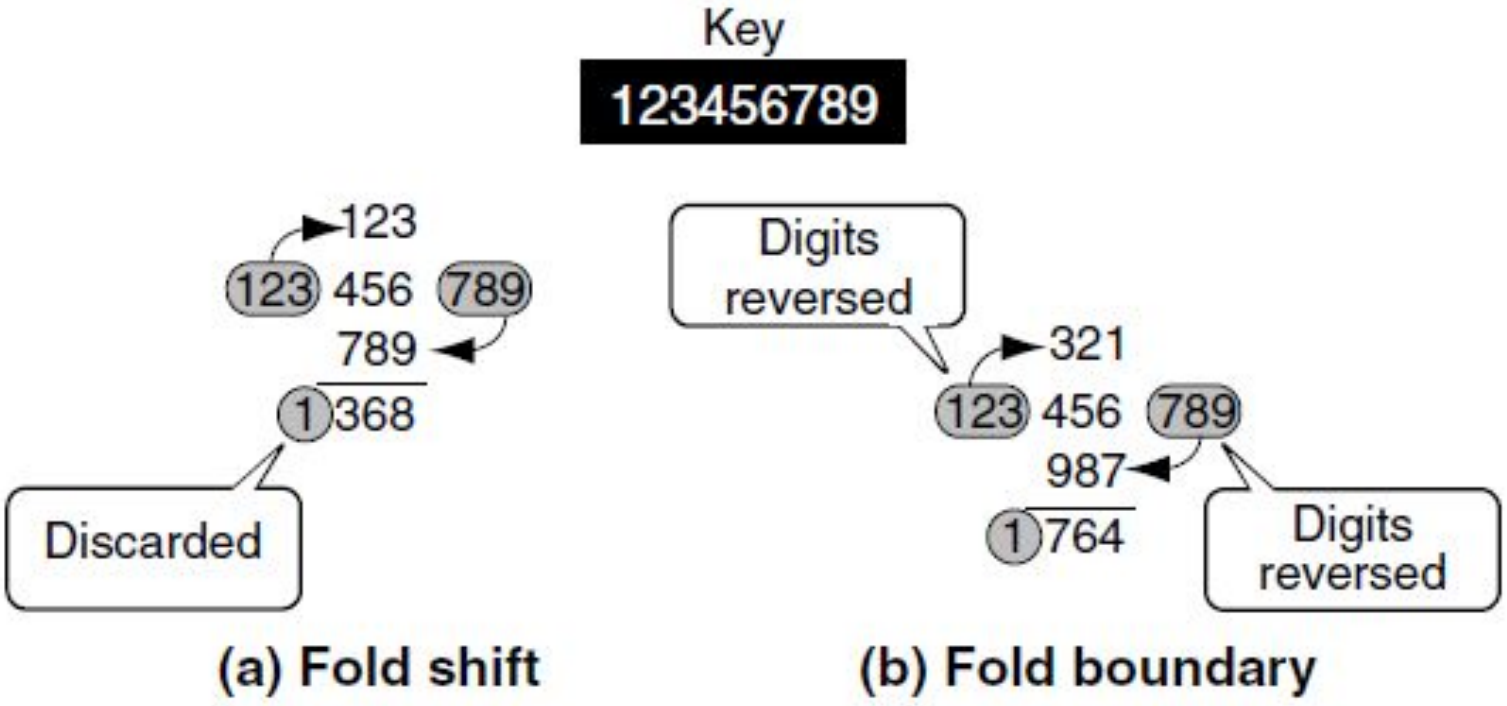
Folding Methods

Two folding methods are used

- ✔ fold shift
- ✔ fold boundary

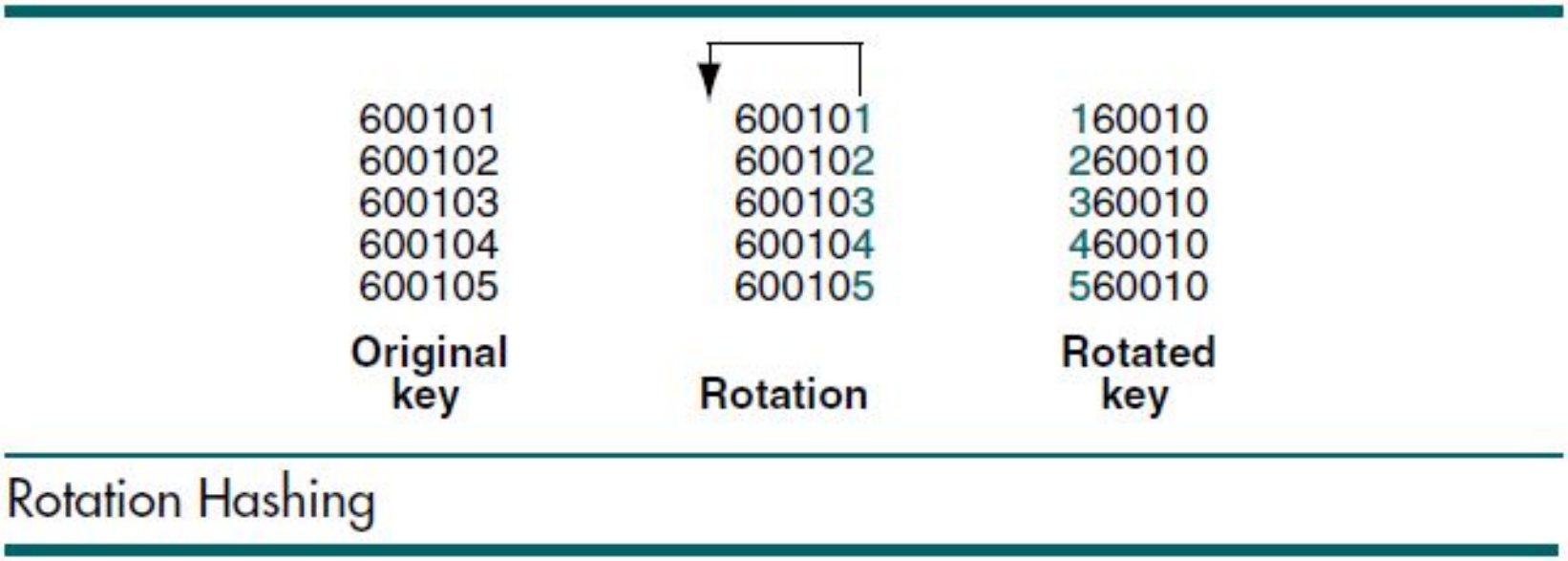
In fold shift the key value is divided into parts whose size matches the size of the required address. Then the left and right parts are shifted and added with the middle part.

Hash Fold Examples



Rotation Method

- Is incorporated in combination with other hashing methods



Rotation Hashing

Pseudorandom Hashing

In pseudorandom hashing the key is used as the seed in a pseudorandom-number generator, and the resulting random number is then scaled into the possible address range using modulo-division

$y = ax + c$

Perfect Hash Function

- A hash function that maps each item into a unique slot (no collisions).

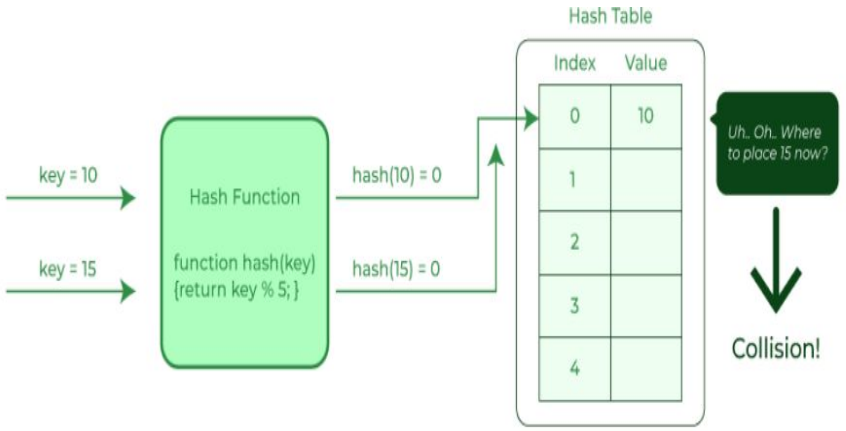
Collision

- The hashing process generates a small number for a big key,
- possibility that two keys could produce the same value.
- situation where the newly inserted key maps to an already occupied, and it must be handled using collision handling technology.

- When one or more hash values compete with a single hash table slot, collisions occur.

Solution

- The next available empty slot is assigned to the current hash value



24, 19, 32, 44,58
k mod 6
identify collisions

24	0
19	1
32//44	2
	3
58	4
	5

Collision Resolution

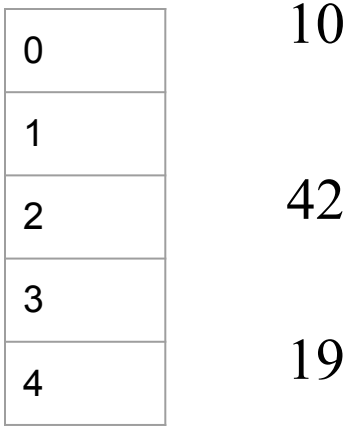
Chaining ----(open hashing
Open addressing:---closed hashing
- Linear Probe
- Quadratic probe
- Double hashing

chaining--open hashing

- This technique implements a linked list and is the most popular collision resolution techniques

42, 19,10,12 ;; k mod 5

Maximum length of the chain, minimum length of the chain



Load factor(alpha)=total number of keys/ total number of slots
=4/5

Advantages---

1. easy insertion, deletion as per linked list operations
2. simple

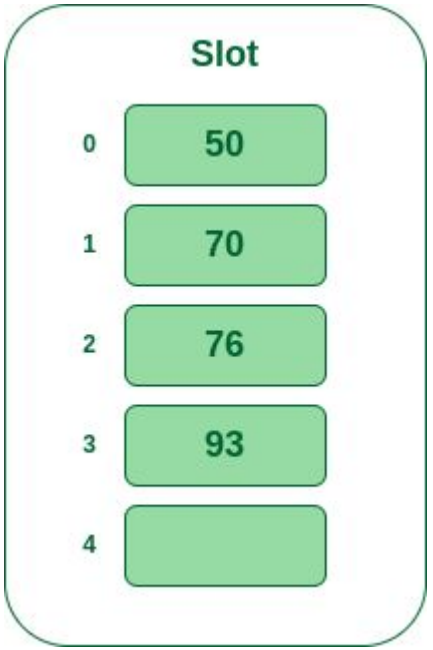
disadvantages:::

more search time when the linked list is long--O(n)

uses extra space

Linear probing

- –the next probe interval is fixed to 1.
- It supports best caching at cluster
- consider a simple hash function as “key mod 5”
- keys that are to be inserted are 50, 70, 76, 93.



keys, 43,135,72,23,99,19,82
linear probing
hash function k mod 10

19,#,72,43,23,135,82,#,#,99,
max probe---5 for key---82

search : O(n) as worst case ;else O(1)
Deletion:: suppose 43 deleted
Primary clustering
secondary clustering

Keys 1,3,12,4,25,6,18,20,8 are inserted into empty hash table of length 10 using open addressing with hash function i^2 mod 10 using linear probing. find the resultant hash table and find the maximum probe value.

20,1,8,#,12,25,4,6,18,3
MAximum probe--9for the key 8

Advantages of linear probing are as follows –

- Linear probing requires very less memory.
- It is less complex and is simpler to implement.

Disadvantages of linear probing are as follows –

- Linear probing causes a scenario called "primary clustering" in which there are large blocks of occupied cells within the hash table.
- The values in linear probing tend to cluster which makes the probe sequence longer and lengthier.

Quadratic probing

- the probe distance is calculated based on the quadratic equation. This is considerably a better option as it balances clustering and caching.
- H(K)=k mod 10
- H'(K)= H(K)+i^2 mod 10
- 42,16,91,33,18,27,36,62

36 mod 10=6
6+1^2 mod 10=7
6+2^2 mod 10=0

62 mod 10=2
2+1^2 mod 10=3
2+2^2 mod 10=6
2+3^2 mod 10=1
2+4^2 mod 10=8
2+5^2 mod 10=7
2+6^2 mod 10=8
2+7^2 mod 10=1

The advantages of quadratic probing is as follows −

Quadratic probing is less likely to have the problem of primary clustering and is easier to implement than Double Hashing.

The disadvantages of quadratic probing are as follows −

Quadratic probing has secondary clustering. This occurs when 2 keys hash to the same location, they have the same probe sequence. So, it may take many attempts before an insertion is being made.

no guarantee to find a slot for keys

Double hashing

- Double hashing uses a hash function of the form (h1(k) + i*h2(k))mod m where h1 and h2 are auxiliary hash functions and m is the size of the hash table.
- h1(k)=k mod 11 h2(k)=8-(k mod 8) ; h1(k)+i.h2(k) mod 11

keys 20,34,45,70,56

20 mod 11----9
34 mod 11--1
45 mod 11--1, 8-(45 mod 8)=8-5=3, 1+3 mod 11=4
70 mod 11--4, 8-(70 mod 8)=2, 4+2 mod 11=6

56 mod 11=1, 8-(56 mod 8)=8, 1 +1.8 mod 11=9
1+2.8 mod 11=6
1+3.8 mod 11 =3

#, 34, #,56,45,#,
70,#,#,20,#

he advantage of double hashing is as follows –

- no extra space
- no primary clustering and secondary clustering
- Double hashing finally overcomes the problems of the clustering issue.

The disadvantages of double hashing are as follows:

- Double hashing is more difficult to implement than any other.

Theorem. If h is chosen from a universal class of hash functions and is used to hash n keys into a table of size m, where $n \leq m$, the expected number of collisions involving a particular key x is less than 1.

How can we create a set of universal hash functions? One possibility is as follows:

1. Choose the table size m to be prime.
2. Decompose the key x into $r + 1$ "bytes" so that $x = \langle x_0, x_1, \ldots, x_r \rangle$, where the maximal value of any x_i is less than m.
3. Let $a = \langle a_0, a_1, \ldots, a_r \rangle$ denote a sequence of $r + 1$ elements chosen randomly such that $a_i \in \{0, 1, \ldots, m - 1\}$. There are m^{r+1} possible such sequences.
4. Define a hash function h_a with $h_a(x) = \sum_{i=0}^{r} a_i x_i \mod m$.
5. $\mathcal{H} = \bigcup_a \{h_a\}$ with m^{r+1} members, one for each possible sequence a.

Consider double hashing of the form
h(k,i)=(h 1 (k)+ih 2 (k)) mod m
Where h 1 (k)=k mod m
h 2 (k)=1+(k mod n)
Where n=m-1and m=701
for k=123456, what is the difference between first and second probes in terms of slots?

=> h(k, i) = (h1(k)+ih2 (k)) mod m
=> Where h1(k)=k mod m,
h2 (k)=1+(k mod n)
n=m-1,
m=701
k = 123456

Now,

=> h1(k) = 123456 mod 701 = 80
=> h2(k) = 1 + (123456 mod 700) = 1 + 256 = 257

1st probe: when i =1

=> h(k, i) = h1(k) + ih2(k)
=> h(k, 1) = h1(k) + h2(k) = 80 + 257 = 337

2nd probe: when i =2

=> h(k,2) = h1(k) + 2*h2(k)
= 80 + 2*257

=> h(k,2) = 80 + 514 = 594

So, difference between first two probes = 594 – 337
= 257

Consider a hash table of size seven, with starting index zero, and a hash function (7x+3) mod 4. Assuming the hash table is initially empty, which of the following is the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing ? Here "___" denotes an empty location in the table.

i ➥ 3, 10, 1, 8,__ ,____,___,

ii ➥ 1, 3, 8, 10,___,___,___.

iii ➥ 1,___,3,___, 8,,10

iv ➥ 3,10,___,___, 8,___,___.

option--1

The hash function used in double hashing is of the form:

i ➥ $h(k, i) = (h_1(k) + h_2(k)$

ii ➥ $h(k, i) = (h_1(k) + h_2(k) - i) \bmod m$

iii ➥ $h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$

iv ➥ $h(k, i) = (h_1(k) - i\, h_2(k)) \bmod m$

Option C

Consider the hash table of size 11 that uses open addressing with linear probing. Let h(k) = k mod 11 be the hash function. A sequence of records with keys 43, 36, 92, 87, 11, 47, 11, 13, 14 is inserted into an initially empty hash table. The bins of which are indexed from 0 to 10. What is the index of the bin into which the last record is inserted?

7

A hash function h defined h(key)=key mod 7, with linear probing, is used to insert the keys 44, 45, 79, 55, 91, 18, 63 into a table indexed from 0 to 6. What will be the location of key 18 ?

5

Linear Search/ Search Algorithm (sequential search)

LA - linear array with N elts; Algorithm to find an element with a value of ITEM, and return its position

1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop

The original array elements are : (searching for 5)
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3

How many comparisons will happen in the following code (next slide) , for **worst case**, for an array of N elements?

Number of element comparisons----------n
Number of index comparisons-----n+1
Total comparisons-----2N+1

```
//linear search function
int linearSearch(int array[],int target, int size){
    int position=0;
    while(array[position]!=target){
        if(position==size)
            return -1;
        else  position++;}
return position;  }

// code snippet of main()
int p=linearSearch(arr,t,s);
      if  (p==-1)
           printf("%d is not in the array",t);
    else printf("%d is found at %d position",t,p);
```

Sentinel Search:

Can we reduce the number of comparisons linear search

HOW???

//last element of the array is replaced with the element to be searched

//The number of comparisons in the worst case here will be (N +2).

(N+2)<(2N+1)

SENTINEL SEARCH ANALYSIS

Number of element comparisons----------N+1 (as one element inserted)
Number of index comparisons-----1
Total comparisons-----N+2

```
// linear search
    for (int i = 0; i < length; i++)
{
        if (array[i] ==
elementToSearch) {
            return i;
        }
    }
    return -1;
```

```
// sentinel search
   i=0;
Array[n]=elementTosearch;
    while (array[i] != elementToSearch) {
        i++;
    }
  if( (i < N-1)
{
    printf(" Item Found “);
}
else
{
    printf( " Item Not Found“);
}
```

Example:

Input: *arr[] = {10, 20, 180, 30, 60, 50, 110, 100, 70}, x = 180*

Output: *180 is present at index 2*

Input: *arr[] = {10, 20, 180, 30, 60, 50, 110, 100, 70}, x = 90*

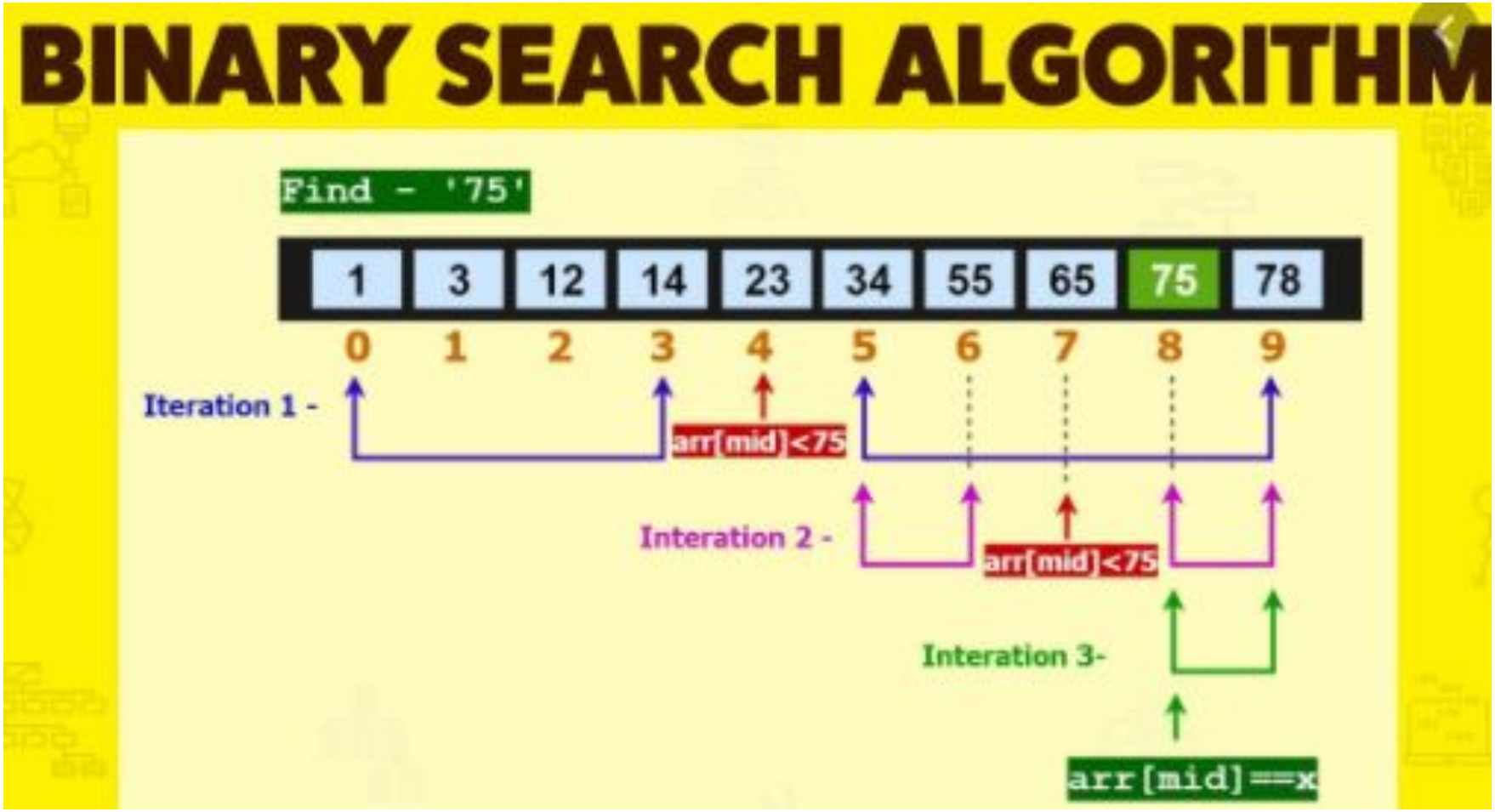
Output: *Not found*

Binary Search

Binary Search:

//Searches a sorted array by repeatedly dividing the search interval in half
The process starts **from the middle** of the input array:

If the target = element in the middle, **return its index**.
Else If the target > element in the middle,
search the right half repeatedly.
Else search the left half repeatedly.


BINARY SEARCH ALGORITHM
Find - '75'
1
3
12
14
23
34
55
65
75
78
0
1
2
3
4
5
6
7
8
9
Iteration 1 -
arr[mid]<75
Interation 2 -
arr[mid]<75
Interation 3-
arr[mid]==x

BinSearch(a,10,18)

```
1   Algorithm BinSearch(a, n, x)
2   // Given an array a[1 : n] of elements in nondecreasing
3   // order, n ≥ 0, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       low := 1; high := n;
7       while (low ≤ high) do
8       {
9           mid := ⌊(low + high)/2⌋;
10          if (x < a[mid]) then high := mid − 1;
11          else if (x > a[mid]) then low := mid + 1;
12               else return mid;
13      }
14      return 0;
15  }
```

Low	High	Mid	condition
1	10	5	1<=10
6			18<10, 18>10
		8	6<=10
			18<16, 18>16
9			9<=10
		9	18<18, 18>18

Algorithm 3.3 Iterative binary search

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20

X=18

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

Linear search	Binary search
It follows a sequential approach.	This follows a divide and conquer approach.
It works well on unsorted data.	To apply binary search, data has to be sorted.
Its worst-case time complexity is O(n).	Its worst-case time complexity is O(log n)
It is slower in comparison to binary search.	It is more efficient than linear search.
In linear search, we compare an element with every other element.	In binary search, we don't compare an element with all other elements. We leave a few comparisons.
We prefer linear search only for small-sized data.	It is preferred for large-sized data.
We can implement linear search on all linear data structures like arrays and linked lists.	We can implement binary search only on those data structures that permit 2-way traversal.

Key Operations for Abstract Strings

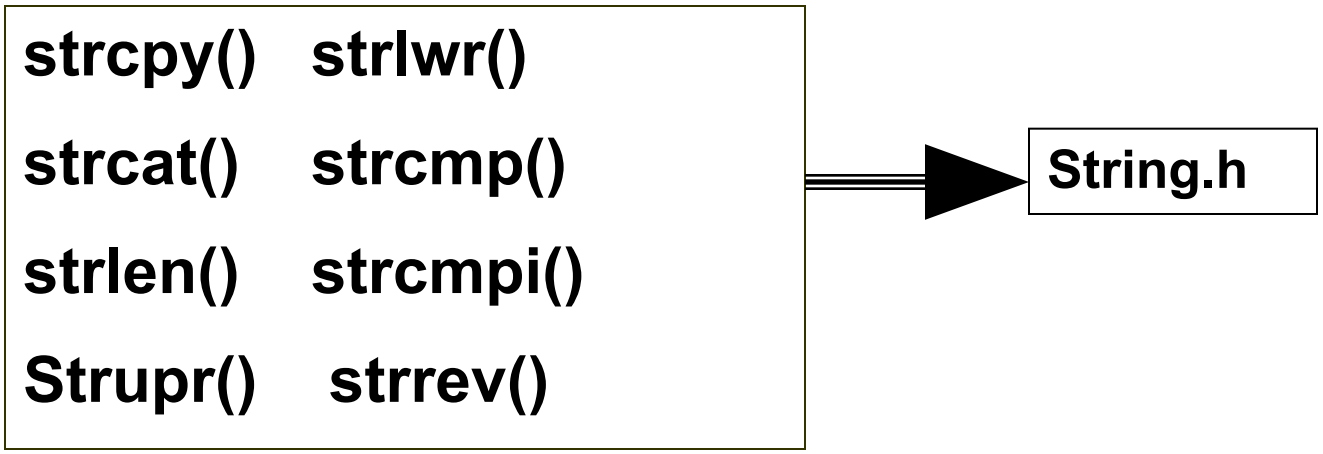
The concatenation of two strings
- places a second string at the end of the first string;

The extraction of substrings,
Searching a string for a matching substring
- determines if the given string is contained within the main string

Matching regular expressions
- searching for patterns (starting / ending with “a” , does not contain “a”, etc.,)

String

- ✔ It can be defined as a group characters which is terminated by a NULL
- ✔ Each character of a string occupies one byte and last character of a string is always the character '\0'
- ✔ \0 □ Null character and it stands for a character with a value of zero



String ADT:

An Abstract Data Type (ADT) consists of a set of values, a defined set of properties of these values, and a set of operations for processing the values.

String ADT Values:

- assumed to have information coded not only in the individual characters (alphabets, numbers and special characters) but also in the ordering of the characters. (meaningful names, eids, product codes etc)

Defined set of Properties

The component characters are from the ASCII character set
The characters are comparable
They have a length, from 0 to n

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.
2)	strcpy(destination, source)	copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	compares the first string with second string. If both strings are same, it returns 0.
5)	strrev(string)	returns reverse string.
6)	strlwr(string)	returns string characters in lowercase.
7)	strupr(string)	returns string characters in uppercase.

Strcpy() function :

It copies the contents of one string into another string.

Syntax : strcpy(string1,string2);

```
#include<stdio.h>
#include<string.h>
void main()
{
char str[25],cpy[25];
printf("\n Enter a String");
gets(str);
strcpy(cpy,str);
printf("\n The source string is %s",str);
printf("\n The copied string is %s",cpy);
}
```

OUTPUT

Enter a String : CSC

The Source string is : CSC

The Copied string is : CSC

Strcat() function :

it concatenates the source string at the end of the target string

Syntax : strcat(string1,string2);

```
#include<stdio.h>
#include<string.h>
void main()
{
char str[25],str1[25];
printf("\n Enter a String");
gets(str);
printf("\n Enter another String");
gets(str1);
printf("\n The concatenated string is %s",strcat(str,str1));
}
```

OUTPUT

Enter a String : CSC

Enter another String : COMPUTER

The Concatenated string is : CSC COMPUTER

Strcmp() function :

It compares two strings to find whether the strings are equal or not.

Syntax : strcmp(string1,string2);

```
#include<stdio.h>
#include<string.h>
void main()
{
char str[25],str1[25];
int x;
printf("\n Enter a String");
gets(str);
printf("\n Enter another String");
gets(str1);
x=strcmp(str,str1);
If(x==0)
printf(“\n Strings are equal”);
else if(x>0)
printf("\n The string1 %s is greater than string2 %s”,str,str1);
else
printf("\n The string2 %s is greater than string1 %s”,str1,str);
}
```

OUTPUT

```
Enter a String : JAIKUMAR
Enter another String : SASIKUMAR
The string2 SASIKUMAR is greater than string1 JAIKUMAR
```

Strcmpi() function :

It compares two strings without regard to case to find whether the strings are equal or not. (i ignorecase)

Syntax : strcmpi(string1,string2);

```
#include<stdio.h>
#include<string.h>
void main()
{
char str[25],str1[25];
int x;
printf("\n Enter a String");
gets(str);
printf("\n Enter another String");
gets(str1);
X=strcmpi(str,str1);
if(x==0)
printf(“\n The two Strings are equal”);
else if(x>0)
printf("\n The string1 %s is greater than string2 %s”,str,str1);
else
printf("\n The string2 %s is greater than string1 %s”,str1,str);
}
```

OUTPUT

```
Enter a String : ABC
Enter another String : abc
The Two strings are equal
```

strrev() function :

used to reverse a string. It takes only one argument.

Syntax : strrev(string);

```
#include<stdio.h>
#include<string.h>
void main()
{
char str[25];
printf("\n Enter a String");
gets(str);
printf("\n The Reversed string is %s",strrev(str));
}
```

OUTPUT

Enter a String : SHIVA

The reversed string is : AVIHS

strupr() function :

used to convert a string to uppercase. It takes only one argument.

Syntax : **strupr(string);**

```
#include<stdio.h>
#include<string.h>
void main()
{
char str[25];
printf("\n Enter a String");
gets(str);
printf("\n The case changed  string is %s",strupr(str));
}
```

OUTPUT

Enter a String : csc

The case changed string is : CSC

strlwr() function :

used to convert a string to Lowercase. It takes only one argument.

Syntax : **strlwr(string);**

```
#include<stdio.h>
#include<string.h>
void main()
{
char str[25];
printf("\n Enter a String");
gets(str);
printf("\n The case changed  string is %s",strlwr(str));
}
```

OUTPUT

Enter a String : CSC

The case changed string is : csc

strlen() function :

used to count the number of characters present in a string. It takes only one argument.

Syntax : **int variablename=strlen(string);**

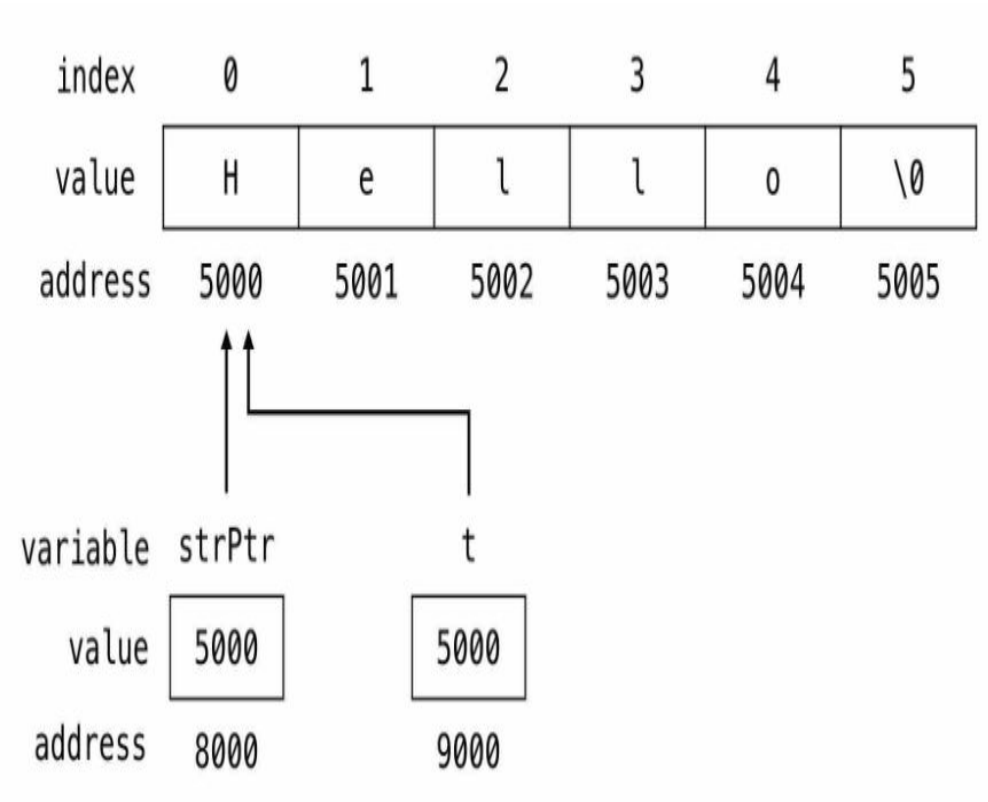
```
#include<stdio.h>
#include<string.h>
void main()
{
char str[25];
printf("\n Enter a String");
gets(str);
printf("\n The length of the  string is %d",strlen(str));
}
```

Enter a String : ABC

The length of the string is : 3

Strings and Pointers

```
#include <stdio.h>
 main() {
 // pointer variable to store string
  char *strPtr = "Hello";
  // temporary pointer variable
  char *t = strPtr;
    // print the string
  while(*t != '\0') {
    printf("%c", *t);
  // move the t pointer to
//the next memory location
    t++;
  }
  }
```



SUBSTRING

A **substring** is itself a string that is part of a longer string

str={“this is a test string”}

test
Substring

examples

- "*program*" is a substring of "*programmer*"
- "*ram*" is a substring of "*programmer*"
- "" (empty string) is a substring of "*programmer*"

operations on substring

- extraction
  //Extracts the mentioned number of characters from the given string
  Inputs needed???

- Index
  //returns the position of the substring in the string

DEMO

SUBSTRING EXTRACTION
(starting position & Number of characters --length)

SUBSTRING EXTRACTION USING POINTERS
(starting position & ending position)

SUBSTRING FUNCTIONS IN <string.h>----index related operations in substring

strchr ( )	Returns pointer to first occurrence of char in str1
strrchr ( )	last occurrence of given character in a string is found
strstr ( )	Returns pointer to first occurrence of str2 in str1
strncat()	concatenates n characters
strncpy()	copies n characters

strncpy

Description

The C library function char *strncpy(char *dest, const char *src, size_t n) copies up to n characters from the string pointed to, by src to dest. In a case where the length of src is less than that of n, the remainder of dest will be padded with null bytes.

Declaration

Following is the declaration for strncpy() function.

```
char *strncpy(char *dest, const char *src, size_t n)
```

Parameters

- dest − This is the pointer to the destination array where the content is to be copied.
- src − This is the string to be copied.
- n − The number of characters to be copied from source.

Return Value

This function returns the final copy of the copied string.

demo----strncpy.c

strncat()

Description

The C library function char *strncat(char *dest, const char *src, size_t n) appends the string pointed to by src to the end of the string pointed to by dest up to n characters long.

Declaration

Following is the declaration for strncat() function.

```
char *strncat(char *dest, const char *src, size_t n)
```

Parameters

- dest − This is pointer to the destination array, which should contain a C string, and should be large enough to contain the concatenated resulting string which includes the additional null-character.
- src − This is the string to be appended.
- n − This is the maximum number of characters to be appended.

Return Value

This function returns a pointer to the resulting string dest.

demo----strncat.c

strchr()

The function strchr() searches the occurrence of a specified character in the given string and returns the pointer to it.

```
char *strchr(const char *str, int ch)
```

str – The string in which the character is searched.

ch – The character that is searched in the string str.

Return Value of strchr()

It returns the pointer to the first occurrence of the character in the given string, which means that if we display the string value of the pointer then it should display the part of the input string starting from the first occurrence of the specified character.

demo----strchr.c

strrchr()

The **strrchr() function** searches the last occurrence of the specified character in the given string. This function works quite opposite to the function strchr() which searches the first occurrence of the character in the string.

```
char *strrchr(const char *str, int ch)
```

str – The string in which the character ch is searched.

ch – The character that needs to be searched

Return value of strrchr()

It returns the pointer to the last occurrence of the character in the string. Which means if we display the return value of the strrchr() then it should display the part of the string starting from the last occurrence of the specified character.

demo----strrchr.c

strstr()

The **strstr() function** searches the given string in the specified main string and returns the pointer to the first occurrence of the given string.

```
char *strstr(const char *str, const char *searchString)
```

str – The string to be searched.

searchString – The string that we need to search in string str

Return value of strstr()

This function returns the pointer to the first occurrence of the given string, which means if we print the return value of this function, it should display the part of the main string, starting from the given string till the end of main string.

demo----strstr.c

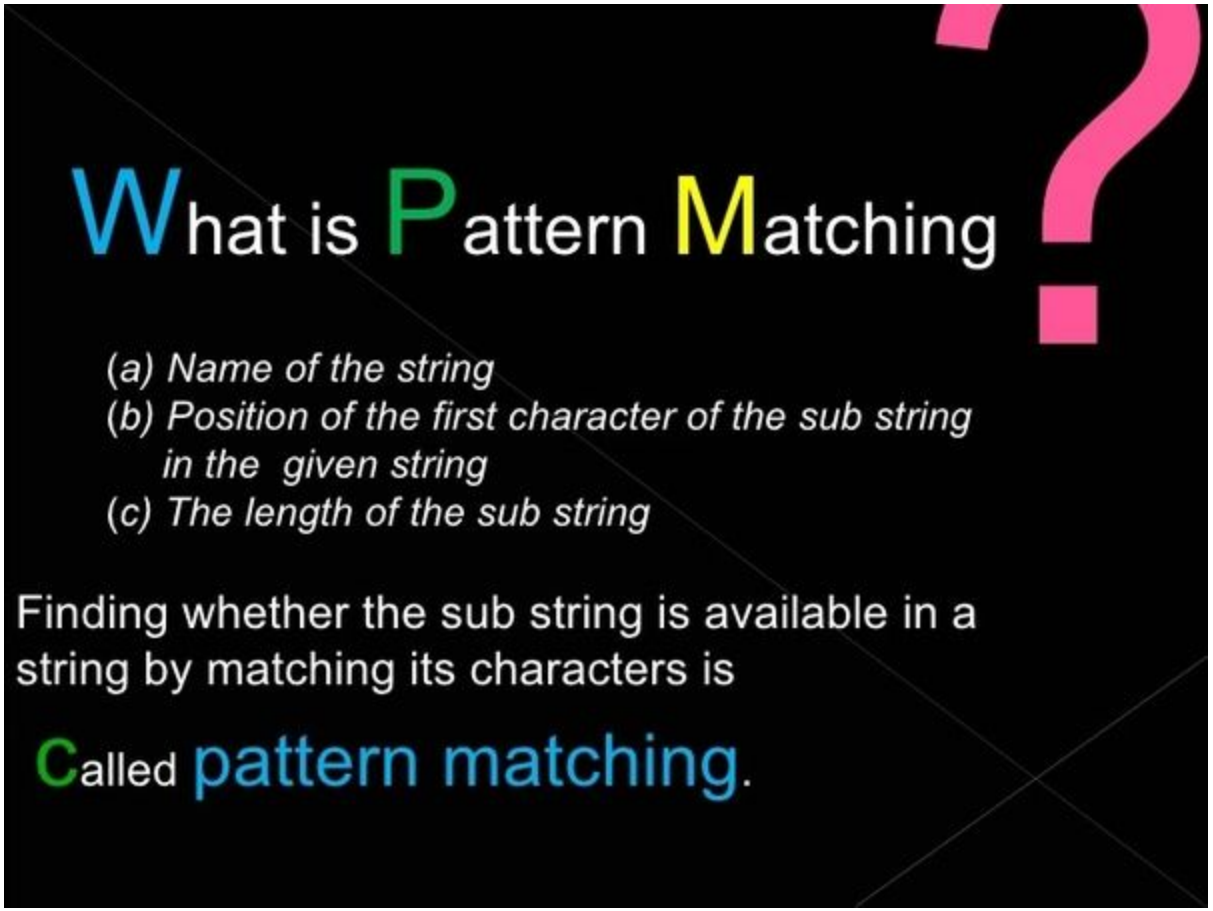
Drawbacks of strstr()

1. When the text and pattern are same, strstr will display only one index

2. Also, based on the requirements, pattern matching output cannot be customized in strstr

3. Efficiency can be improved by reducing the number of comparisons

PATTERN MATCHING



APPLICATIONS

1. Compilation process to check the syntax
2. FIND & REPLACE
3. EMAIL VALIDATION
4. Plagiarism Detection
5. Spelling Checker
6. Search engines or content search in large databases

IMPLEMENTATION….

- Regular programming languages make use of regular expressions

- ALGORITHMS

Algorithms

- Naive Pattern matching
- KMP Algorithm
- Rabin-Karp Algorithm
- Finite Automata
- Boyer Moore Algorithm

NAIVE PATTERN MATCHING

txt

A	B	C	A	B	A	\0
0	1	2	3	4	5	6

Pat

C	A	B	\0
0	1	2	3

txt

A	B	C	A	B	A	\0
0	1	2	3	4	5	6

Pat

C	A	B	\0
0	1	2	3

txt

A	B	C	A	B	A	\0
0	1	2	3	4	5	6

Pat

C	A	B	\0
0	1	2	3

txt

A	B	C	A	B	A	\0
0	1	2	3	4	5	6

Pat

C	A	B	\0
0	1	2	3

txt

A	B	C	A	B	A	\0
0	1	2	3	4	5	6

Pat

C	A	B	\0
0	1	2	3

txt

A	B	C	A	B	A	\0
0	1	2	3	4	5	6

Pat

C	A	B	\0
0	1	2	3

Characteristics

1. Number of iterations=n-m+1
2. Number of comparisons =m*(n-m+1)
3. Time Complexity==O(m*(n-m+1)

Improvements

1. Pattern length is greater than the string'
2. checking the first and/or last characters of pat and text before we compare the remaining characters

CHECKING THE LAST INDICES

txt

A	B	C	A	B	A	\0
0	1	2	3	4	5	6

Pat

C	A	B	\0
0	1	2	3

txt

A	B	C	A	B	A	\0
0	1	2	3	4	5	6

Pat

C	A	B	\0
0	1	2	3

txt

A	B	C	A	B	A	\0
0	1	2	3	4	5	6

Pat

C	A	B	\0
0	1	2	3

txt

A	B	C	A	B	A	\0
0	1	2	3	4	5	6

Pat

C	A	B	\0
0	1	2	3

txt

A	B	C	A	B	A	\0
0	1	2	3	4	5	6

Pat

C	A	B	\0
0	1	2	3

txt

A	B	C	A	B	A	\0
0	1	2	3	4	5	6

Pat

C	A	B	\0
0	1	2	3

PATTERN MATCHING (by checking the end indices first)

```
int mfind(char *string, char *pat)
{
// match the last character of the pattern first and then match from the beginning
int i,j,start=0;
int lasts=strlen(string)-1;
lastp=strlen(pat)-1;
int endmatch=lastp;
for(i=0;endmatch<=lasts;endmatch++,start++)
{
if string[endmatch]==pat[lastp])
for (j=0,i=start;j<lastp &&string[i]==pat[j];i++,j++)
;
if(j==lastp)
return start;
}
return-1;
}
```

DRAWBACKS OF NAIVE PATTERN MATCHING

- Repeatedly checking the characters in the input string
- No preprocessing is done for the pattern

BEST CASE

The best case occurs in the general –NAÏVE PATTERN matching algorithm is when the first character of the pattern is not present in text at all.

txt[] = "AABCCAADDEE";
pat[] = "FAA";

worst case

The worst case of Naive Pattern Searching occurs in following scenarios with maximum number of comparisons.
1) When all characters of the text and pattern are same.

txt[] = "AAAAAAAAAAAAAAAAAA";
pat[] = "AAAAA";

2) Worst case also occurs when only the last character is different.

txt[] = "AAAAAAAAAAAAAAAAAB";
pat[] = "AAAAB";

KMP algorithm

KNUTH --MORRIS--PRATT Algorithm

preprocessing is done for the pattern

Complexity is O(m+n)

failure function

```
void fail(char *pat)
{   // compute the pattern's failure function
int n=strlen(pat);
failure[0]=-1;
for (j=1; j <n;j++)
i=failure[j-1];
while((pat[j]!=pat[i+1]) && i>==0)
i=failure[i];
if (pat[j]==pat[i+1])
failure[j]=i+1;
else
failure[j]=-1;
}
}
```

KMP algorithm

```
int pmatch(char *string, char * pat)
{ //Kunth,Morris, pratt algorithm
int i=0,j=0;
int lens=strlen(string);
int lenp=strlen(pat);
while(i<lens  && j< lenp) {
if string[i]==pat[j]){
i++;j++;}
else if (j==0) i++;
else j=failure[j-1]+1;
}
return ((j==lenp) ?(i-lenp):-1);
}
```