Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Applications – Stack and Queue

Stack

- Parenthesis balancing [lexical analysis]
- Expression conversion and evaluation
- Function calls
- Used in advanced data structures for traversing

- **Expression conversion and evaluation:**
  - Infix to postfix
  - Infix to prefix
  - Postfix evaluation
  - Prefix evaluation

Expression : combination of operands and operators
1. Infix : a + b (human)
2. Prefix : + a b } (computer/machine)
3. Postfix : a b + } ↳ CPU → ALU

Queue

- Jobs submitted to printer [spooler Directory]
- In Network setups – file access of file server machine is given to First come First serve basis
- Calls are placed on a queue when all operators are busy
- Used in advanced data structures to give efficiency.
- Process waiting queues in OS

Postfix Evaluation

- Process each element of postfix expression from left to right
- If element is operand
  - Push it on a stack
- If element is operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op2 – first popped element
    - Op1 – second popped element
  - Perform current element (Operator) operation between Op1 and Op2
  - Again push back result onto the stack
- When single value will remain on stack, it is final result
- e.g. 4 5 6 * 3 / + 9 + 7 -

Postfix evaluation

Postfix expression : 4 5 6 * 3 / + 9 + 7 -

L ⟶ R

Result = 16

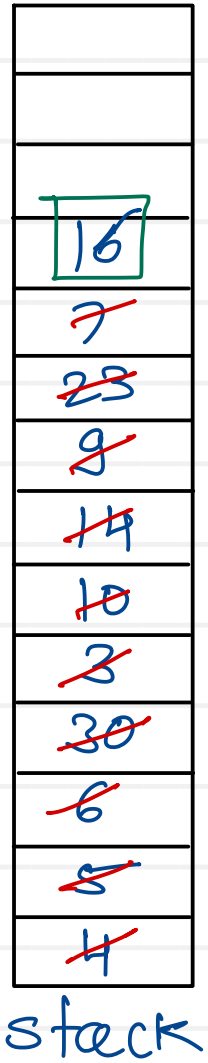
⑤ 23 − 7 = 16

④ 14 + 9 = 23

③ 4 + 10 = 14

② 30 / 3 = 10

① 5 * 6 = 30



Prefix Evaluation

- Process each element of prefix expression from right to left
- If element is operand
  - Push it on a stack
- If element is operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op1 – first popped element
    - Op2 – second popped element
  - Perform current element (Operator) operation between Op1 and Op2
  - Again push back result onto the stack
- When single value will remain on stack, it is final result
- e.g. - + + 4 / * 5 6 3 9 7

Prefix evaluation

Prefix expression : - + + 4 / * 5 6 3 9 7

l ← r

Result = 16

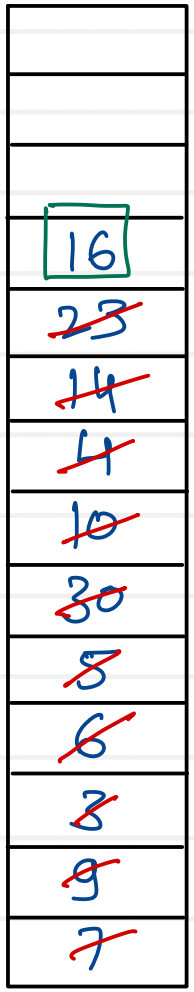
⑤ 23 − 7 = 16

④ 14 + 9 = 23

③ 4 + 10 = 14

② 30 / 3 = 10

① 5 * 6 = 30



infix = 10 + 20

postfix = "10 20 +"

String arr[] = postfix.split(' ');

arr = {"10", "20", "+"};

use parseInt() to convert string into integer

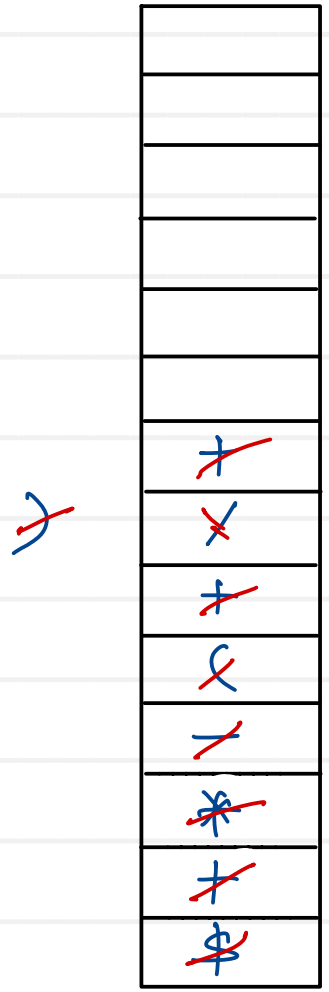
Infix to Postfix Conversion

- Process each element of infix expression from left to right
- If element is Operand
  - Append it to the postfix expression
- If element is Operator
  - If priority of topmost element (Operator) of stack is greater or equal to current element (Operator), pop topmost element from stack and append it to postfix expression
  - Repeat above step if required
  - Push element on stack
- Pop all remaining elements (Operators) from stack one by one and append them into the postfix expression
- e.g. a * b / c * d + e – f * h + i

Infix to Postfix conversion

Infix expression : 1 $ 9 + 3 * 4 - (6 + 8 / 2) + 7

Postfix expression : 19$34*+682/+-7+



Infix to Prefix Conversion

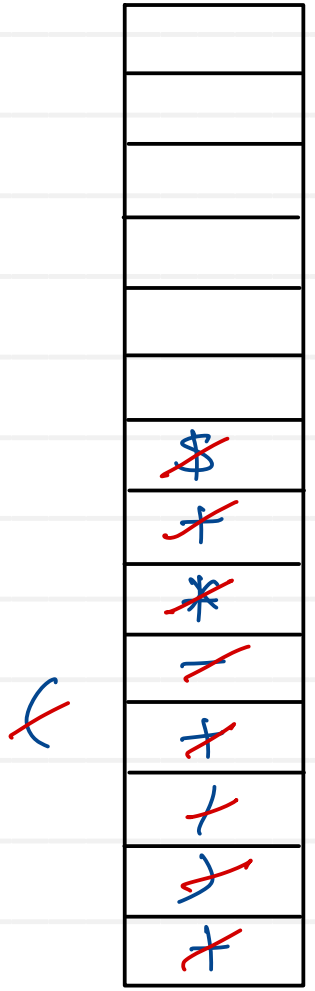
- Process each element of infix expression from right to left
- If element is Operand
  - Append it to the prefix expression
- If element is Operator
  - If priority of topmost element of stack is greater than current element (Operator), pop topmost element from stack and append it to prefix expression
  - Repeat above step if required
  - Push element on stack
- Pop all remaining elements (Operators) from stack one by one and append them into the prefix expression
- Reverse prefix expression
- e.g. a * b / c * d + e – f * h + i

Infix to Prefix conversion

Infix expression : 1 $ 9 + 3 * 4 - (6 + 8 / 2) + 7

Expression : 7 2 8 / 6 + 4 3 * 9 1 $ + - +

Prefix expression : + - + $ 1 9 * 3 4 + 6 / 8 2 7



Prefix to Postfix

- Process each element of prefix expression from right to left
- If element is an Operand
  - Push it on to the stack
- If element is an Operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op1 – first popped element
    - Op2 – second popped element
  - Form a string by concatenating Op1, Op2 and Opr (element)
  - String = “Op1+Op2+Opr”, push back on to the stack
- Repeat above two steps until end of prefix expression.
- Last remaining on the stack is postfix expression
- e.g. * + a b – c d

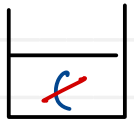
Postfix to Infix

- Process each element of postfix expression from left to right
- If element is an Operand
  - Push it on to the stack
- If element is an Operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op2 – first popped element
    - Op1 – second popped element
  - Form a string by concatenating Op1, Opr (element) and Op2
  - String = “Op1+Opr+Op2”, push back on to the stack
- Repeat above two steps until end of postfix expression.
- Last remaining on the stack is infix expression
- E.g. a b c - + d e – f g – h + / *

Valid Parantheses

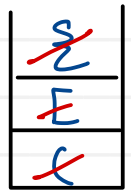
Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
An input string is valid if:
- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

Example 1:
Input: s = "()"
Output: true



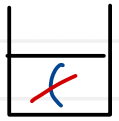
) == (

Example 2:
Input: s = "()[]{}"
Output: true



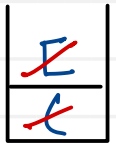
) == (
] == [
} == {

Example 3:
Input: s = "(]"
Output: false



] != (

Example 4:
Input: s = "([])"
Output: true



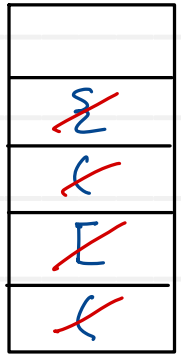
] == [
) == (

1. Create stack to push brackets
2. traverse string from left to right
   2.1 if bracket is opening
       then push it on stack
   2.2 if bracket is closing
       2.2.1 if stack is empty, return false.
       2.2.2 if stack is not empty,
           - pop one bracket from stack,
           - if they are matching, continue
           - if they are not matching, return false.
3. if stack is not empty, return false
4. if stack is empty, return true

Parenthesis balancing using stack

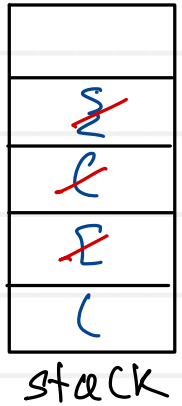
5+([9-4]*(8-{6/2}))

] == [
} == {
) == (
) == (



5+([9-4]*(8-{6/2]))

] == [
} == {
] != (



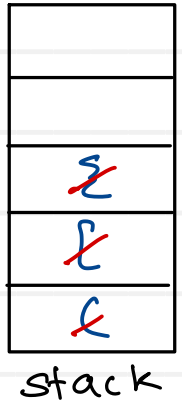
opening ([{
0 1 2

closing)] }
0 1 2

string
↓
indexOf( )
↓
returns index of char
returns -1 if char not found

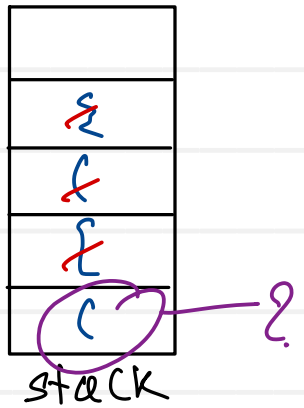
5+([9-4]*8-{6/2}))

] == [
} == {
) == (
) == {



5+([9-4]*(8-{6/2})

] == [
} == {
) == (



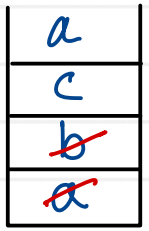
Remove all adjacent duplicates in string

You are given a string s consisting of lowercase English letters. A duplicate removal consists of choosing two adjacent and equal letters and removing them.

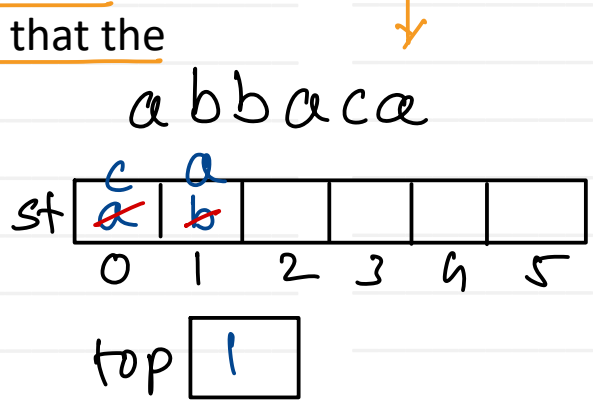
We repeatedly make duplicate removals on s until we no longer can.

Return the final string after all such duplicate removals have been made. It can be proven that the answer is unique.

Example 1:
Input: s = "abbaca"
Output: "ca"



Example 2:
Input: s = "azxxzy"
Output: "ay"




```
string removeDuplicates(string s) {
    int n = s.length();
    char st[] = new char[n];
    int top = -1;

    for(i=0; i<n; i++) {
        char ch = s.charAt(i);
        if(top != -1 && st[top]==ch)
            top--;
        else
            st[++top] = ch;
    }
    return new string(arr, 0, top+1);
}
```

Algorithm

Program : set of rules / instructions to processor / CPU
Algorithm : set of instructions to human (programmer)
- step by step solution of given problem

- Algorithms are programming language independent.
- Algorithms can be written in any human understandable language.
- Algorithms can be used as a templates

Algorithm ⟶ Program / function
(Template) (Implementation)

Find sum of array elements
step 1 : create sum & initialize to 0
step 2 : traverse array for 0 to N-1 index
step 3 : Add every element of array in sum
step 4 : return / print sum

e.g. searching, sorting
↓ ↓
linear / binary selection / bubble / quick

Algorithm analysis

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

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

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

— Notations used to denote time / space complexity

$O()$ (upper bound) $\Omega()$ (lower bound) $\Theta()$ (Avg/tight bound)

Time complexity

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

1. Print 1D array on console

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

No. of iterations of loop = n

time ∝ iterations

time ∝ n

$T(n) = O(n)$

2. Print 2D array on console

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

iterations of outer loop = m

iterations of inner loop = n

total iterations = m * n

Time ∝ m * n

$T(m, n) = O(m*n)$

$m \approx n$

Time ∝ n * n

$T(n) = O(n^2)$

Time complexity

3. Add two numbers

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

- irrespective of input values, time required to execute will be same always
- constant time requirement and it is denoted as

$$T(n) = O(1)$$

4. Print table of given number

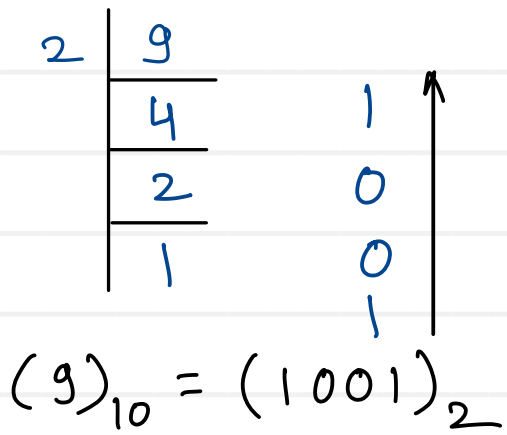
```
void printTable (int num) {
    for(i=1; i<=10; i++)
        sysout(num * i);
}
```

- irrespective of num, loop will iterate fixed (10) times.
- constant time requirement.

$$T(n) = O(1)$$

Time complexity

5. Print binary of decimal number



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

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

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

$n = 9, 4, 2, 1$

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

$= n/2^0, n/2^1, n/2^2, n/2^{itr}$

$1 = n/2^{itr}$

$2^{itr} = n$

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

$itr \log 2 = \log n$

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

$time \propto \frac{1}{\log 2} \log n$

$T(n) = O(\log n)$

Time complexity

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

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

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

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

```
for(i=0; i<n; i++);→n
for(j=0; j<n; j++);→n
```
+ = 2n

Time α 2n
T(n) = O(n)

```
for(i=0; i<n; i++) → n
    for(j=n; j>0; j=j/2)→log n
```
T(n) = O(nlogn)

Time complexity

```
for(i=n/2; i<=n; i++)          -> n
  for(j=1; j+n/2<=n; j++)      -> n
    for(k=2; k<=n; k=k*2)      -> log n
```

Total itr = $n * n * \log n$

$= n^2 \log n$

```
for(i=n/2; i<=n; i++)          -> n
  for(j=1; j<=n; j=2*j)        -> log n
    for(k=1; k<=n, k=k*2)      -> log n
```

total itr = $n * \log n * \log n$

$= n \log^2 n$

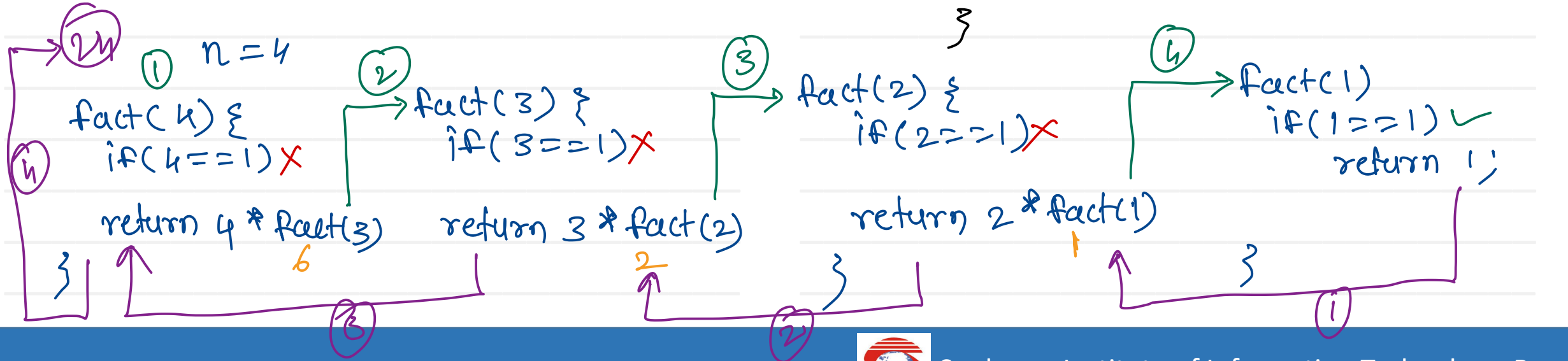
Recursion

- Calling function within itself
- we can use recursion
  - if we know formula/process in terms of itself
  - if we know terminating condition

e.g. $n! = n * (n-1)!$

$0! = 1! = 1$

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



Algorithm analysis

Iterative

- loops are used

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

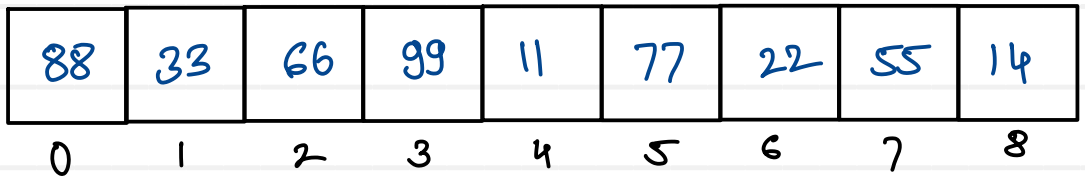
Recursive

- recursion is used

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

Linear search (random data)

1. decide/take key from user
2. traverse collection of data from one end to another
3. compare key with data of collection
   - 3.1 if key is matching
     return index/true
   - 3.2 if key is not matching
     return -1/false



key == arr[i]

77
Key

i = 0, 1, 2, 3, 4, 5
key is found

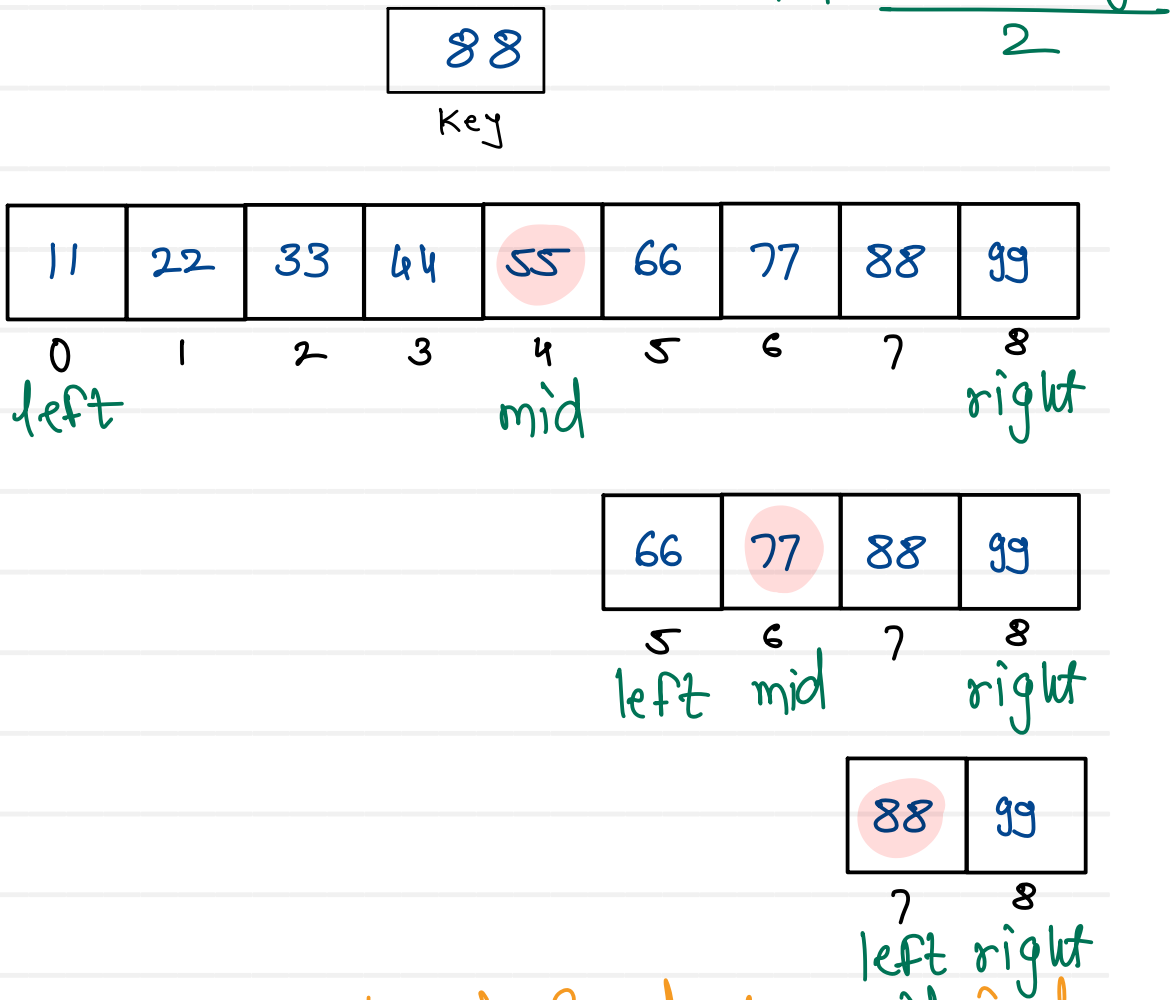
89
Key

i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
key is not found

Binary search

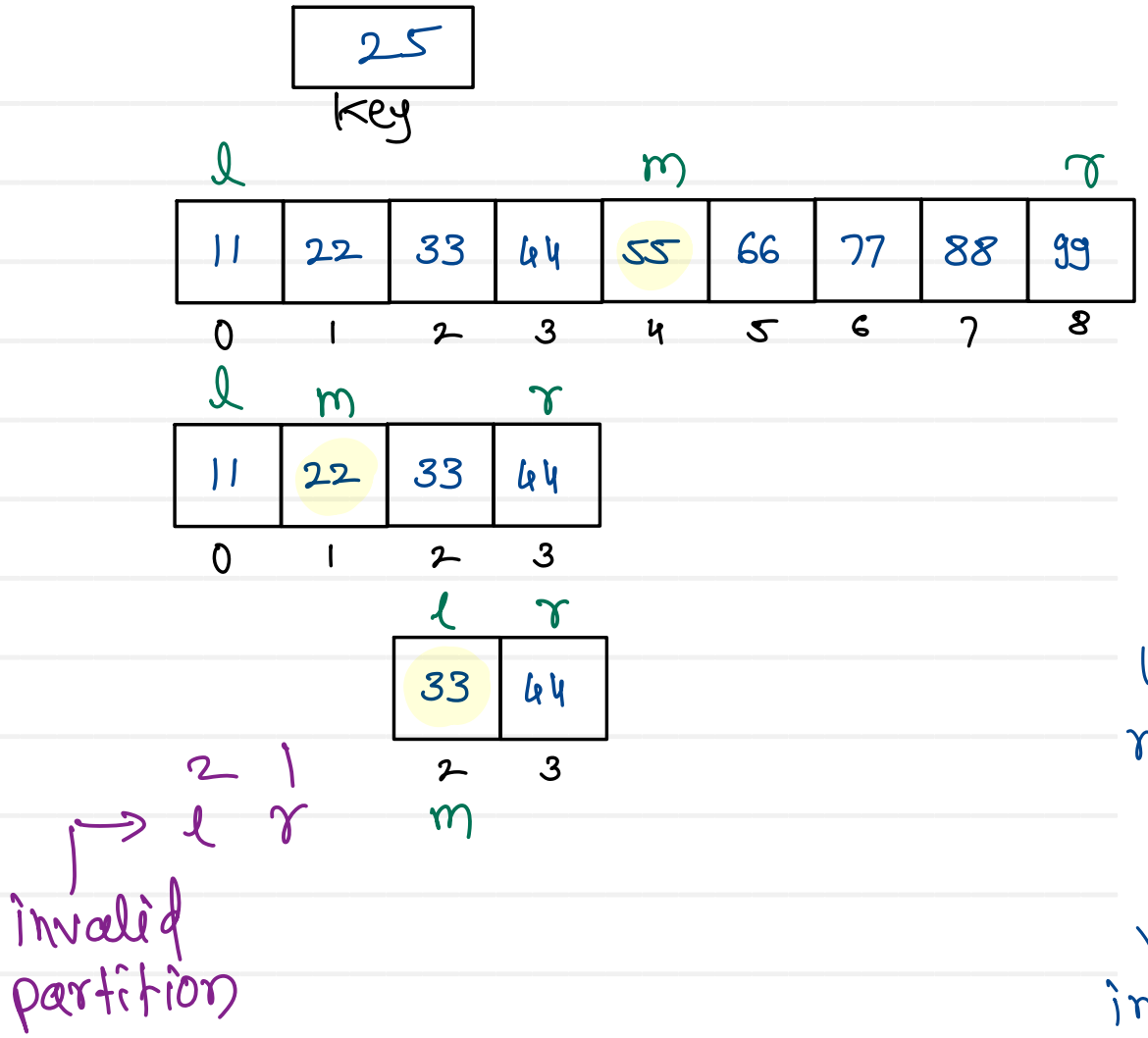
1. take key from user
2. divide array into two parts
   (find middle element)
3. compare middle element with key
3.1 if key is matching
   return index (mid)
3.2 if key is less than middle element
   search key in left partition
3.3 if key is greater than middle element
   search key in right partition
3.4 if key is not matching
   return -1

mid = (left + right) / 2



Key is found at → mid index

Binary search



partition : left → right
left partition : left → mid-1
right partition : mid+1 → right

valid partition : left <= right
invalid partition : left > right

```
l=0 , r=8 , m;
while(l <= r) {
    m = (l+r) /2;
    if( key == arr[m]
        return m;
    else if( key < arr[m])
        right = m-1;
    else
        left = m+1;
}
return -1;
```

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

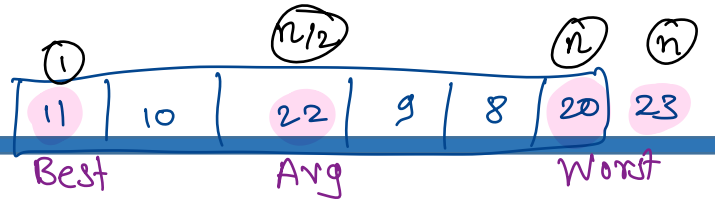
key = 88

l	r	l <= r	m
0	8	T	4
5	8	T	6
7	8	T	7

key = 25

0	8	T	4
0	3	T	1
2	3	T	2
2	1	F	

Searching algorithms analysis



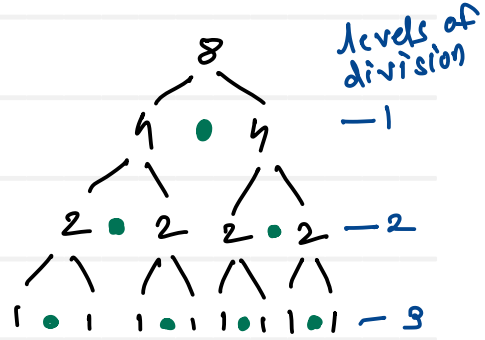
- Time is directly proportional to number of comparisons
- For searching and sorting algorithms, count number of comparisons done

1. Linear search
   - Best case - if key is found at few initial locations → O(1)
   - Average case - if key is found at middle locations → O(n)
   - Worst case - if key is found at last few locations / key is not found → O(n)

S(n)=O(1)

2. Binary search
   - Best case - if key is found at first few levels → O(1)
   - Average case - if key is found at middle levels → O(log n)
   - Worst case - if key is found at last level / not found → O(log n)

S(n)=O(1)



$2^3 = 8 \Rightarrow 2^l = n$ $\quad l = \frac{\log n}{\log 2}$

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com