Administrivia

- Instructor: Piyus Kedia
- Lectures: Mon, Fri – 9:00-10:30 AM – C102
- Office hours: Fri – 10:30-11:00 AM – A505, Research Block
- TF: Riti Choudhary (riti@iiitd.ac.in)

Tentative lecture plan

- Introduction to C
- Revision of recursion (Factorial, Fibonacci sequence, Tower of Hanoi)
- Time complexity of algorithms (Big-Oh notation)
- Array (search and sorting algorithms)
- Stacks, queues, and lists
- Trees
- Binary-tree (Binary search trees, AVL trees, B-trees)
- Heap Trees (Priority queues, Heap sort)
- Hash tables
- Sets (Disjoint sets, Union, Find algorithms)
- Graphs (BFS, DFS, Topological sort, Minimum spanning trees, Shortest path, Huffman coding)

Grading components

MidSem Theory	25
MidSem Lab	10
EndSem Theory	25
EndSem Lab	10
Graded Labs (9 to 11)	20
Quizzes/Homework	10

References

- Algorithms: A Creative Approach
  - Udi Manber
- Introduction to Algorithms, Third Edition
  - Thomas H. Cormen. Charles E. Leiserson. Ronald L. Rivest, Clifford Stein
- Data Structures & Algorithms in C++
  - M.T. Goodrich, R. Tamassia, D. Mount
- The C Programming Language. 2nd Edition
  - Brian Kernighan and Dennis Ritchie
- Online C reference: https://users.cs.cf.ac.uk/Dave.Marshall/C/CE.html

Academic dishonesty

- Please refer to the institute’s plagiarism policy to know more
  - https://www.iiitd.ac.in/academics/resources/acad-dishonesty

Intro to C

- Types
  - Every variable in a C program needs a type declaration
    - Unlike Python, in which types are automatically inferred by the compiler
  - Variables of the same type follow some common properties
    - e.g., all the variables of type int store a 4-byte integer value
    - You can use $sizeof(ty)$ to obtain the size of a given type ty

Basic types

'a', 'b', 'c'

TYPE	SIZE (TYPICAL SIZE)	RANGE (TYPICAL SIZE)
char	1	-128 to 127
unsigned char	1	0 to 255
short int	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
int	>=2 (4)	-2,147,483,648 to 2,147,483,647
unsigned int	>=2 (4)	0 to 4,294,967,295
long long int	8	-2^{63} to $2^{63}-1$
unsigned long long	8	0 to $2^{64}-1$
float	4	1.175494e-38 to 3.402823e+38
double	8	2.225074e-308 to 1.797693e+308

Variable declaration

int i = 0;

```
int i;
char c;
float f;

i = 0;
c = 'a';
f = 1.0;
```

Array type

- An array is a collection of elements of the same type

```
int arr[100]; // declaring an array of 100 elements
// the first element is indexed using 0, second is 1,
// and so on ..
a[2] = 20; // updating the 3rd element
a[139] = 2; // allowed but the behavior is undefined
// it might corrupt the values of the other variables
```

One-dimensional array

```
int arr[10]; // 1-d array
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

x x+4 x+8 . . .

Two-dimensional array

```
int arr[5][3]; // 2-d array
```

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]
a[3][0]	a[3][1]	a[3][2]
a[4][0]	a[4][1]	a[4][2]

n-dimension array

- Similarly, you can have n-dimensional array

Type

- All variables and functions must be declared before their use
- The function declaration contains the types of arguments and the return value
- Every C program must have the $main$ function

The main routine

```
int main(int argc, const char *argv[]) {
    return 0;
}

int main() {
    return 0;
}
```

Hello world!

```
int main(){
    printf(“Hello World!”);
    // printf prints to console
    return 0;
}
```

Hello world!

```
int main(){
   printf(“Hello World!”);
   // printf prints to console
   // compiler gives a warning because
   // printf is not declared
   return 0;
}
```

Hello world!

```
#include <stdio.h>
// stdio.h contains a declaration for printf

int main(){
   printf("Hello World!");
   // printf prints to console
   return 0;
}
```

Compiling and running

```
gcc hello.c
./a.out
```

-O3

```
int a=1;
int b=2;
int c=3;
return a+b;
```

```
gcc hello.c -o hello
./hello
```

Loop (for)

```
for (initialization; condition; update) {
   // body of the for loop
}
```

Loop (for)

i=0 , i<5
i=1 , i<5
i=2 , i<5
i=3 , i<5
i=4 , i<5
i=5 , i<5

```
int i;
for (i = 0; i < 5; i += 1) {
  printf("Hello world!");
}
```

The initialization is done only once on entry

The loop body is executed if the condition is satisfied

The update is done after the execution of the loop body

The loop condition is again checked after the update

Loop (while)

```
while (condition) {
   // body of the while loop
}
```

Loop (while)

i = 0 ; i < 5
i = 1 , i < 5
i = 2
i = 3
i = 4
i = 5 , i < 5

```
int i = 0;
while (i < 5) {
  printf("Hello world!");
  i += 1;
}
```

Loop body is executed if the condition is true

Condition is again checked after the execution of the loop body

Conditional

```
if (condition) {
  // if body
}

if (condition) {
  // if body
}
else {
  // else body
}
```

```
if (condition) {
  // if body
}
else if (condition) {
  // else-if body
}
else {
  // else-body
}
```

if (a > 0) {
}

if (a > 0) {
}
a = -4 else {
}

if (a > 0) {
}
else if (a > 5) {
}
else if (a > -10) {
}
else {
}

Input from user

```
int n;
printf("Enter a number\n");
scanf("%d", &n);
printf("The number is: %d\n", n);
```

Function declaration

RETURN TYPE

ARG1 TYPE

ARG2 TYPE

ARG3 TYPE

```
int foo(int arg1[2], char arg2, float arg3);
```

A function must be declared/defined before its use

Function definition

```
int foo(int arg1[2], char arg2, float arg3) {
  // function body
}
```

A function must be declared/defined before its use

Example

```
#include <stdio.h>

int sum(int arr[], int n) {
    int i, s = 0;
    for (i = 0; i < n; i++) {
        s += i; arr[i];
    }
    return s;
}
int main() {
    int arr[5];
    int i, s;
    printf("Enter five numbers\n");
    for (i = 0; i < 5; i++) {
        scanf("%d", &arr[i]);
    }
    s = sum(arr, 5);
    printf("s=%d\n", s);
    return s;
}
```

size for 1st dimension is optional

Summary

```
int i; // variable declaration
int i = 0; // declaration and initialization
int a[5]; // 1-d array declaration
int b[5][5]; // 2-d array declaration
int c[5][5][5]; // 3-d array declaration

a[1] = 20;
b[1][2] = a[1];
c[1][2][3] = b[1][2];
```

Library functions: printf, scanf (stdio.h)

```
for (i = 0; i < 20; i++) {
  // body
}
while (i < 10) {
  // body
}
if (i < 10) {
  // if-body
}
int foo(int arg1[], char arg2, float arg3);
// function declaration

int foo(int arg1[], char arg2, float arg3) {
// function body
}
```

C languages

- You will learn more about C in the labs and tutorials
- In the class, we will discuss some more topics when they are needed

Algorithm

- Algorithm is a well-defined finite sequence of operations that work on a given input to derive the desired output
  - i.e., an algorithm must terminate

What is data structure?

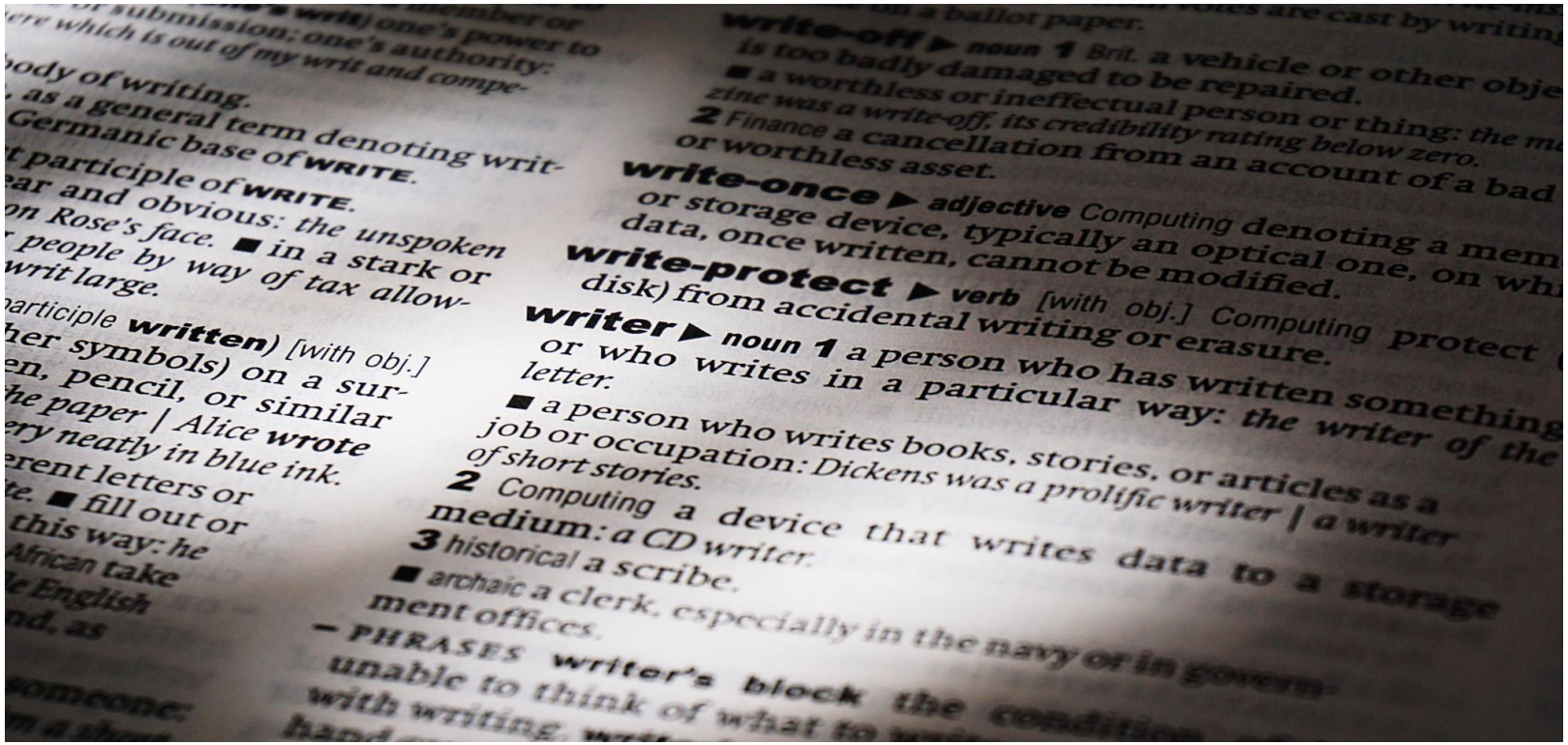
- Data structure is a way of storing and organizing data
  - e.g., array, stack, queue, tree, graph, etc.

Real world examples

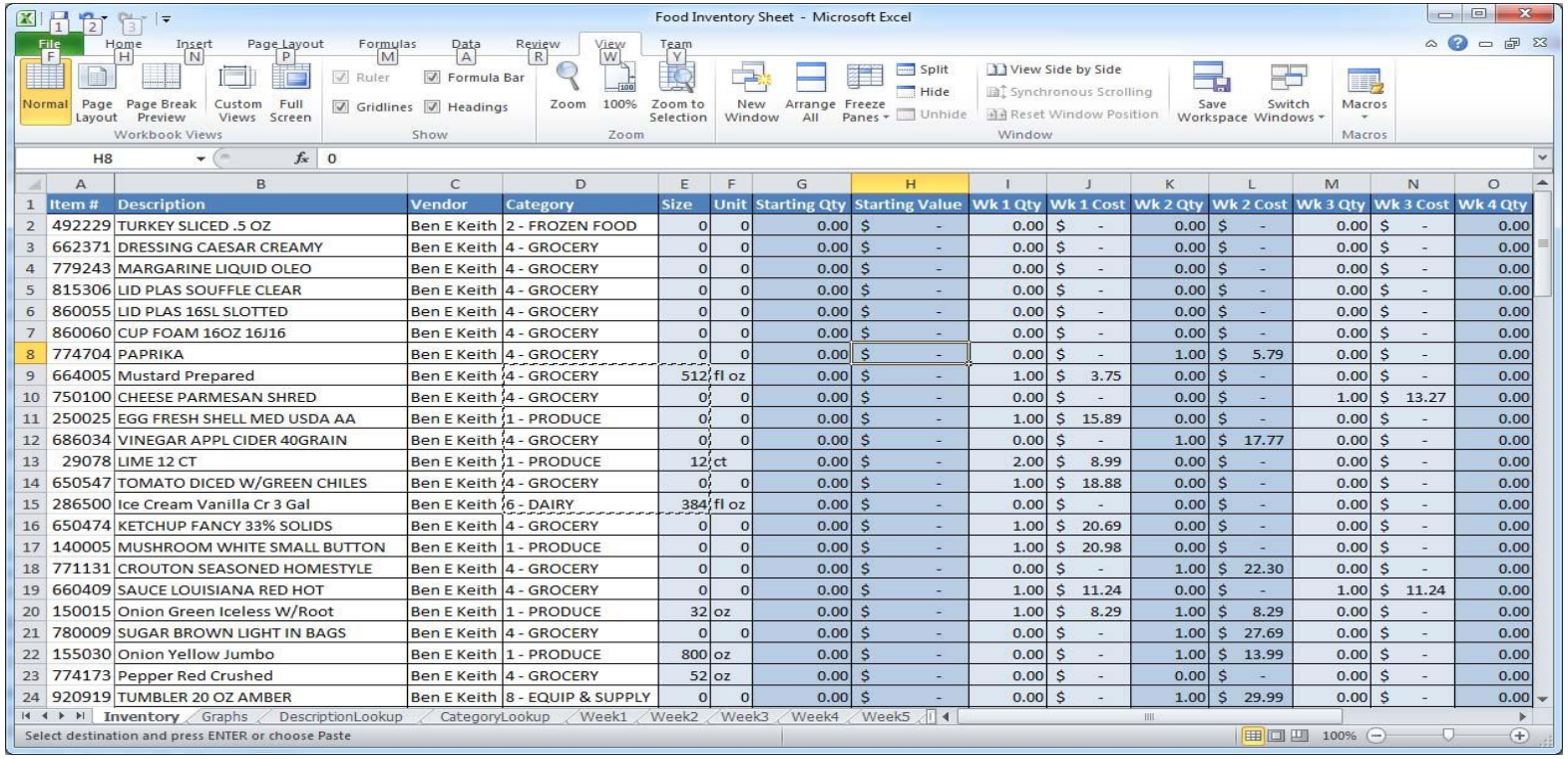
Ticket counter

This Photo by Unknown Author is licensed under CC BY

Dictionary


write-off ▶ noun 1 Brit. a vehicle or other obje
is too badly damaged to be repaired.
■ a worthless or ineffectual person or thing: the m
zine was a write-off; its credibility rating below zero.
2 Finance a cancellation from an account of a bad
or worthless asset.
write-once ▶ adjective Computing denoting a mem
or storage device, typically an optical one, on wh
data, once written, cannot be modified.
write-protect ▶ verb [with obj.] Computing protect
disk) from accidental writing or erasure.
writer ▶ noun 1 a person who has written something
or who writes in a particular way: the writer of the
letter.
■ a person who writes books, stories, or articles as a
job or occupation: Dickens was a prolific writer | a writer
of short stories.
2 Computing a device that writes data to a storage
medium: a CD writer.
3 historical a scribe.
■ archaic a clerk, especially in the navy or in govern-
ment offices.
— PHRASES writer's block the condition
unable to think of what to
with writing.
of submission; one's authority:
ere which is out of my writ and compe-
body of writing.
, as a general term denoting writ-
Germanic base of WRITE.
t participle of WRITE.
ear and obvious: the unspoken
on Rose's face. ■ in a stark or
people by way of tax allow-
writ large.
articple written) [with obj.]
her symbols) on a sur-
en, pencil, or similar
he paper | Alice wrote
ery neatly in blue ink.
erent letters or
te. ■ fill out or
this way: he
African take
e English
nd, as
someone:

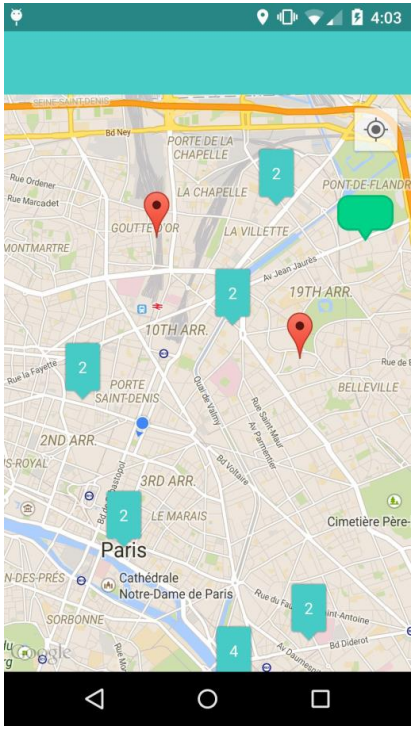
Excel sheet



Item #	Description	Vendor	Category	Size	Unit	Starting Qty	Starting Value	Wk 1 Qty	Wk 1 Cost	Wk 2 Qty	Wk 2 Cost	Wk 3 Qty	Wk 3 Cost	Wk 4 Qty
492229	TURKEY SLICED .5 OZ	Ben E Keith	2 - FROZEN FOOD	0	0	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
662371	DRESSING CAESAR CREAMY	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
779243	MARGARINE LIQUID OLEO	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
815306	LID PLAS SOUFFLE CLEAR	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
860055	LID PLAS 16SL SLOTTED	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
860060	CUP FOAM 16OZ 16J16	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
774704	PAPRIKA	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	1.00	$ 5.79	0.00	$ -	0.00
664005	Mustard Prepared	Ben E Keith	4 - GROCERY	512	fl oz	0.00	$ -	1.00	$ 3.75	0.00	$ -	0.00	$ -	0.00
750100	CHEESE PARMESAN SHRED	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	0.00	$ -	1.00	$ 13.27	0.00
250025	EGG FRESH SHELL MED USDA AA	Ben E Keith	1 - PRODUCE	0	0	0.00	$ -	1.00	$ 15.89	0.00	$ -	0.00	$ -	0.00
686034	VINEGAR APPL CIDER 40GRAIN	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	1.00	$ 17.77	0.00	$ -	0.00
29078	LIME 12 CT	Ben E Keith	1 - PRODUCE	12	ct	0.00	$ -	2.00	$ 8.99	0.00	$ -	0.00	$ -	0.00
650547	TOMATO DICED W/GREEN CHILES	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	1.00	$ 18.88	0.00	$ -	0.00	$ -	0.00
286500	Ice Cream Vanilla Cr 3 Gal	Ben E Keith	6 - DAIRY	384	fl oz	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
650474	KETCHUP FANCY 33% SOLIDS	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	1.00	$ 20.69	0.00	$ -	0.00	$ -	0.00
140005	MUSHROOM WHITE SMALL BUTTON	Ben E Keith	1 - PRODUCE	0	0	0.00	$ -	1.00	$ 20.98	0.00	$ -	0.00	$ -	0.00
771131	CROUTON SEASONED HOMESTYLE	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	1.00	$ 22.30	0.00	$ -	0.00
660409	SAUCE LOUISIANA RED HOT	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	1.00	$ 11.24	0.00	$ -	1.00	$ 11.24	0.00
150015	Onion Green Iceless W/Root	Ben E Keith	1 - PRODUCE	32	oz	0.00	$ -	1.00	$ 8.29	1.00	$ 8.29	0.00	$ -	0.00
780009	SUGAR BROWN LIGHT IN BAGS	Ben E Keith	4 - GROCERY	0	0	0.00	$ -	0.00	$ -	1.00	$ 27.69	0.00	$ -	0.00
155030	Onion Yellow Jumbo	Ben E Keith	1 - PRODUCE	800	oz	0.00	$ -	0.00	$ -	1.00	$ 13.99	0.00	$ -	0.00
774173	Pepper Red Crushed	Ben E Keith	4 - GROCERY	52	oz	0.00	$ -	0.00	$ -	0.00	$ -	0.00	$ -	0.00
920919	TUMBLER 20 OZ AMBER	Ben E Keith	8 - EQUIP & SUPPLY	0	0	0.00	$ -	0.00	$ -	1.00	$ 29.99	0.00	$ -	0.00

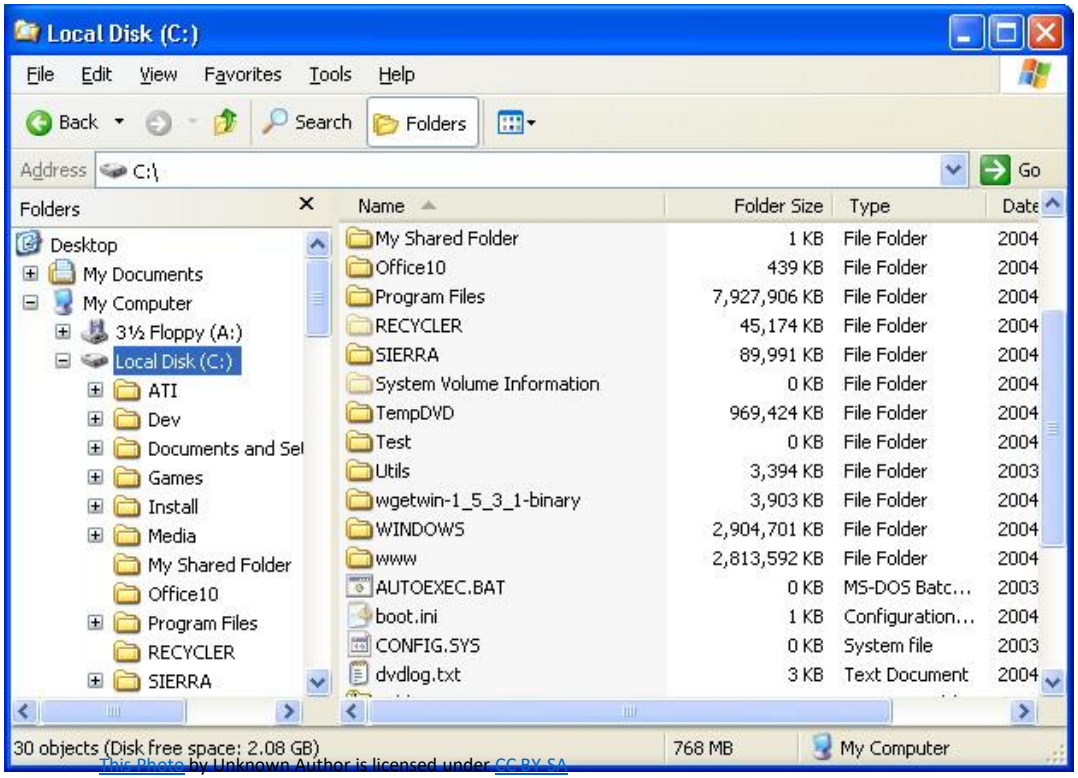
This Photo by Unknown Author is licensed under CC BY-SA

Maps



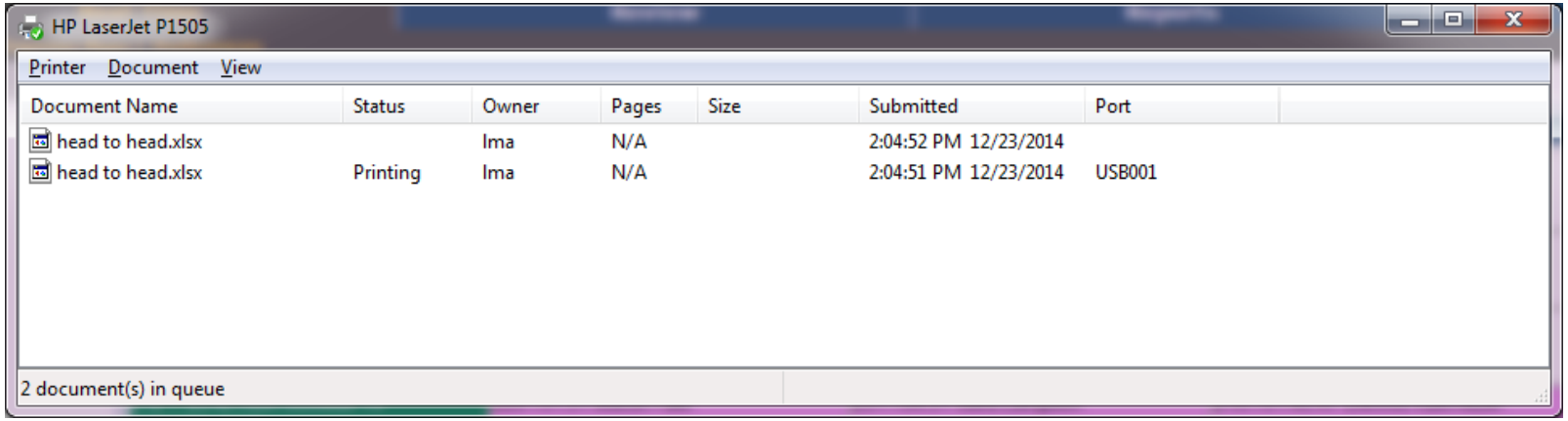
This Photo by Unknown Author is licensed under CC BY-SA

Files and directories



This Photo by Unknown Author is licensed under CC BY-SA

Printer



This Photo by Unknown Author is licensed under CC BY-SA

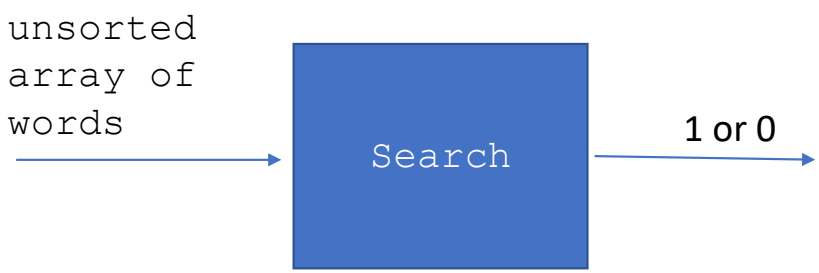
Plates

This Photo by Unknown Author is licensed under CC BY-SA

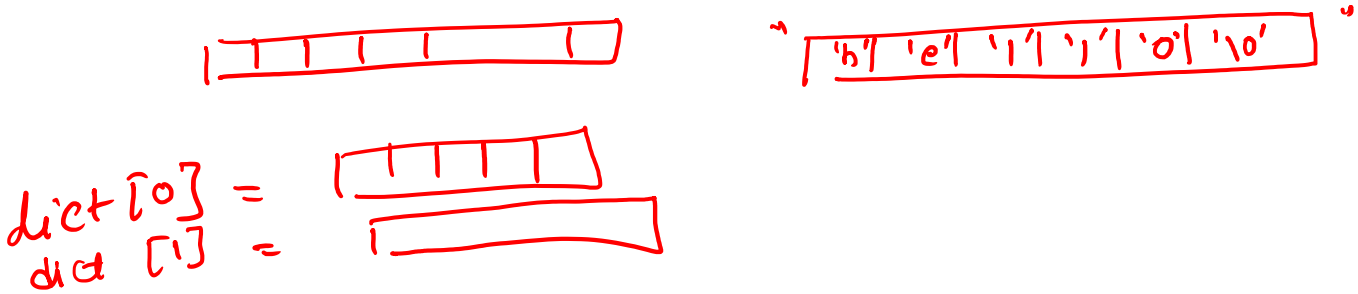
Why do we need different data structures?

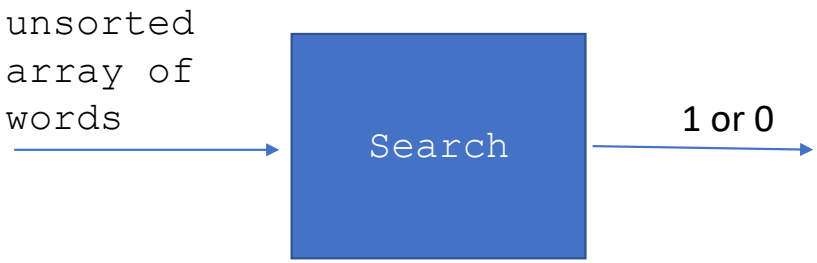
- A single data structure may not suffice for all purposes
- Efficiency of an algorithm depends on the underline data structures

Search



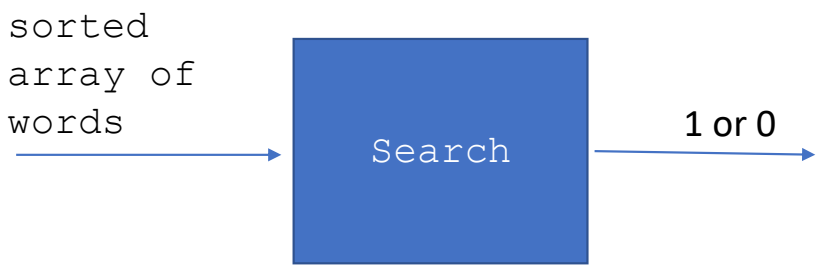
Search






```
#include <string.h>

int search(char dict[][26], int n_words, char word[])
{
   int i;
   for (i = 0; i < n_words; i++) {
      if (strcmp(dict[i], word) == 0) {
         return 1;
      }
   }
   return 0;
}
```

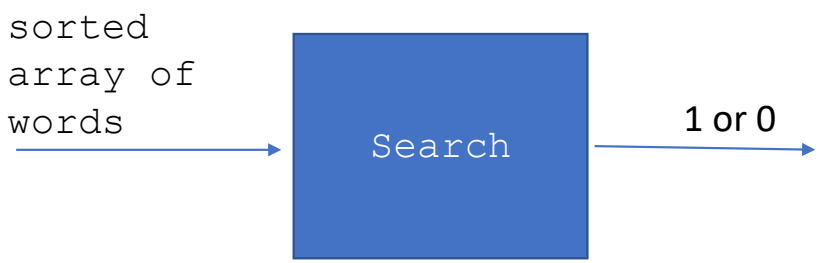
Search



Search

0 = 'a' - 'a'
1 = 'b' - 'a'
2 = 'c' - 'a'
. . .

idx_of[1]
idx_of[2]




```
#include <string.h>

// idx_of[26] contains total number of words
int search(char dict[][26], char word[],
           unsigned int idx_of[27])
{
   unsigned int idx = word[0] - 'a';
   int i;
   assert(idx < 26);
   for (i = idx_of[idx]; i < idx_of[idx+1]; i++) {
      if (strcmp(dict[i], word) == 0) {
         return 1;
      }
   }
   return 0;
}
```

Abstraction

```
#include <stdio.h>

int main() {
  int num;
  printf(“Enter a Number\n”);
  scanf(“%d”, &num);
  printf(“You Entered: %d\n”, num);
  return 0;
}
```

Abstraction

- Abstraction hides the unnecessary implementation details from the users
- For example, a header file in `C` provides a list of function declarations that users can directly use in their program without worrying about the underline implementation
  - e.g., `printf, scanf`, etc., in `stdio.h`

Abstract data type (ADT)

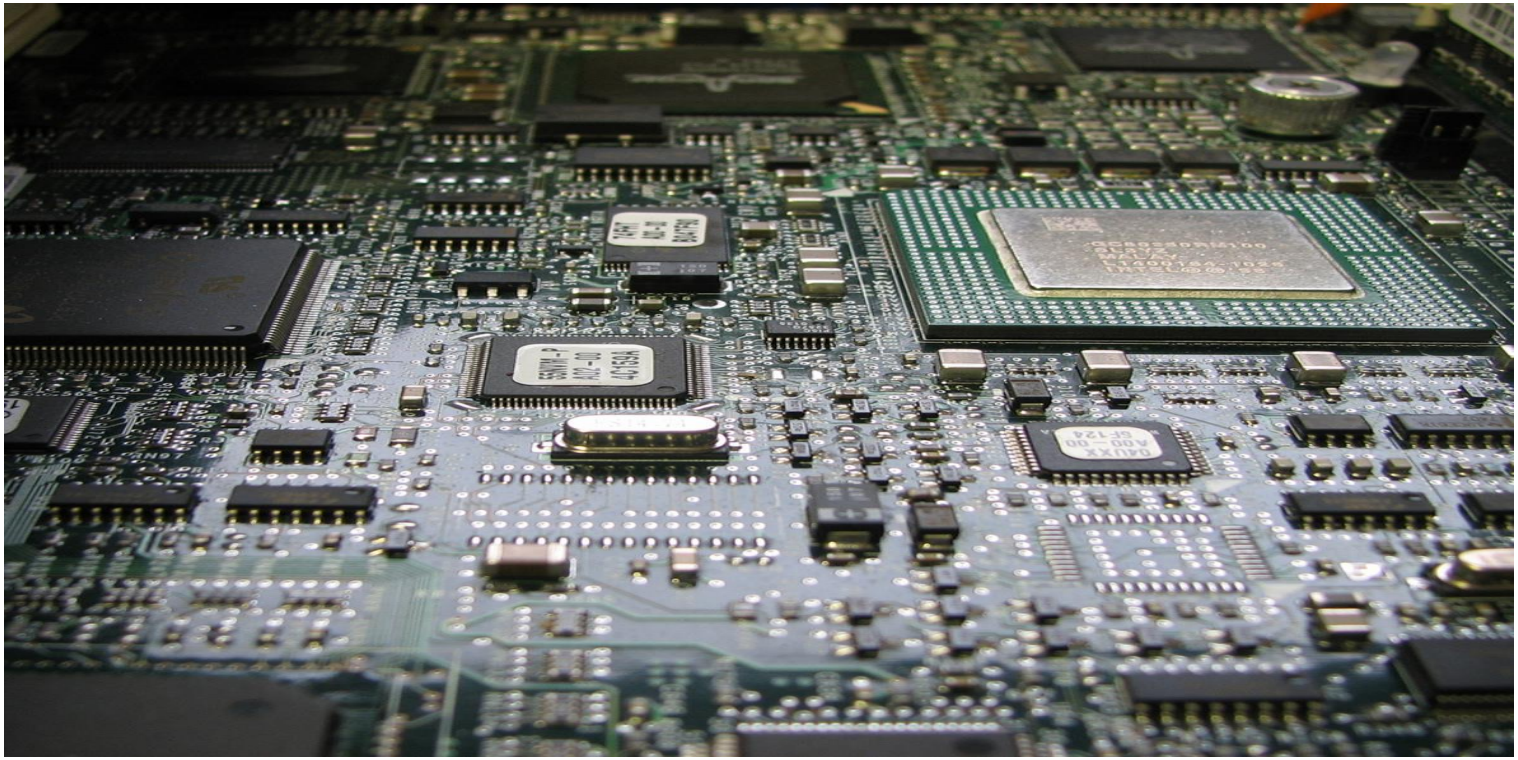
- ADT specifies the user's point of view of a data structure
  - i.e., the supported operations and their semantics, the possible values
- ADT hides the implementation details from the user
- ADT can be defined using a header file in C/C++ or an interface in Java

Abstract (or user’s) view

This Photo by Unknown Author is licensed under CC BY-SA-NC

- Power On/Off
- Select a program
- Start/Pause/Resume

Implementation view



This Photo by Unknown Author is licensed under CC BY-ND

Abstract data type (ADT)

- Dictionary ADT
  - insert => insert a word
  - delete => delete a word
  - search => search a word

Example

- Train ticket booking platform
- Suppose there are thousands of requests, and you can process only one request at a time
  - In which order will you process the request?
  - How do you store the request?

Queue

- Follow first-in, first-out (FIFO) policy
- Queue ADT:
  - QUEUE-EMPTY => returns true if the queue is empty
  - ENQUEUE => insert an item at the end of the queue
  - DEQUEUE => remove an item from the top of the queue

Example

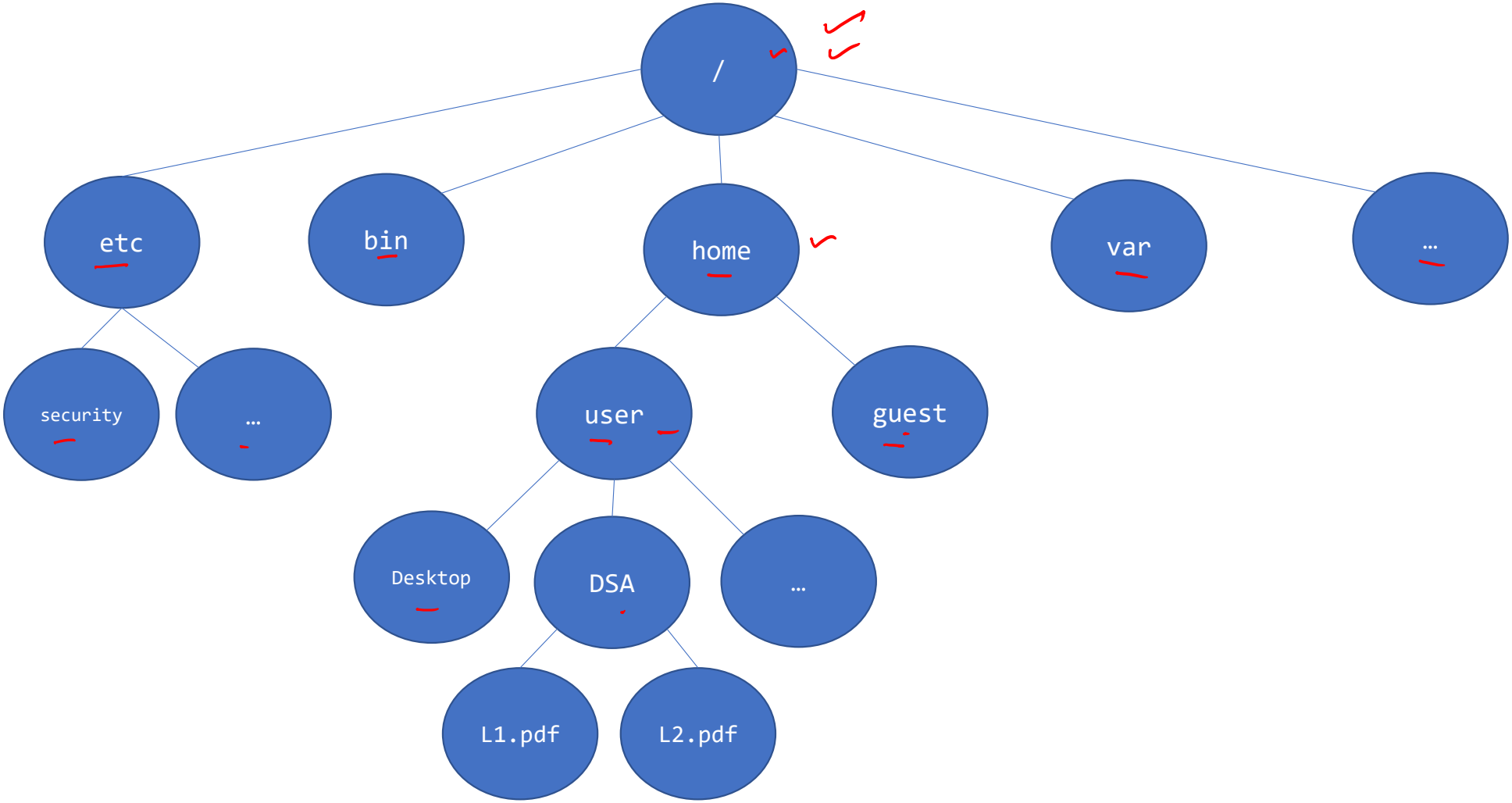
- Undo (ctrl+z) operation in power-point
- How do you store the words to perform undo efficiently?

Stack

- Follow last-in, first-out (LIFO) policy
- Stack ADT:
  - STACK-EMPTY => returns true if the stack is empty
  - PUSH => insert an item on the top of the stack
  - POP => remove an item from the top of the stack

Example

- Files and directories structure in an OS
  - How to efficiently store the relationship between the directory and subdirectories or files?


/
etc
bin
home
var
...
security
...
user
guest
Desktop
DSA
...
L1.pdf
L2.pdf

Tree

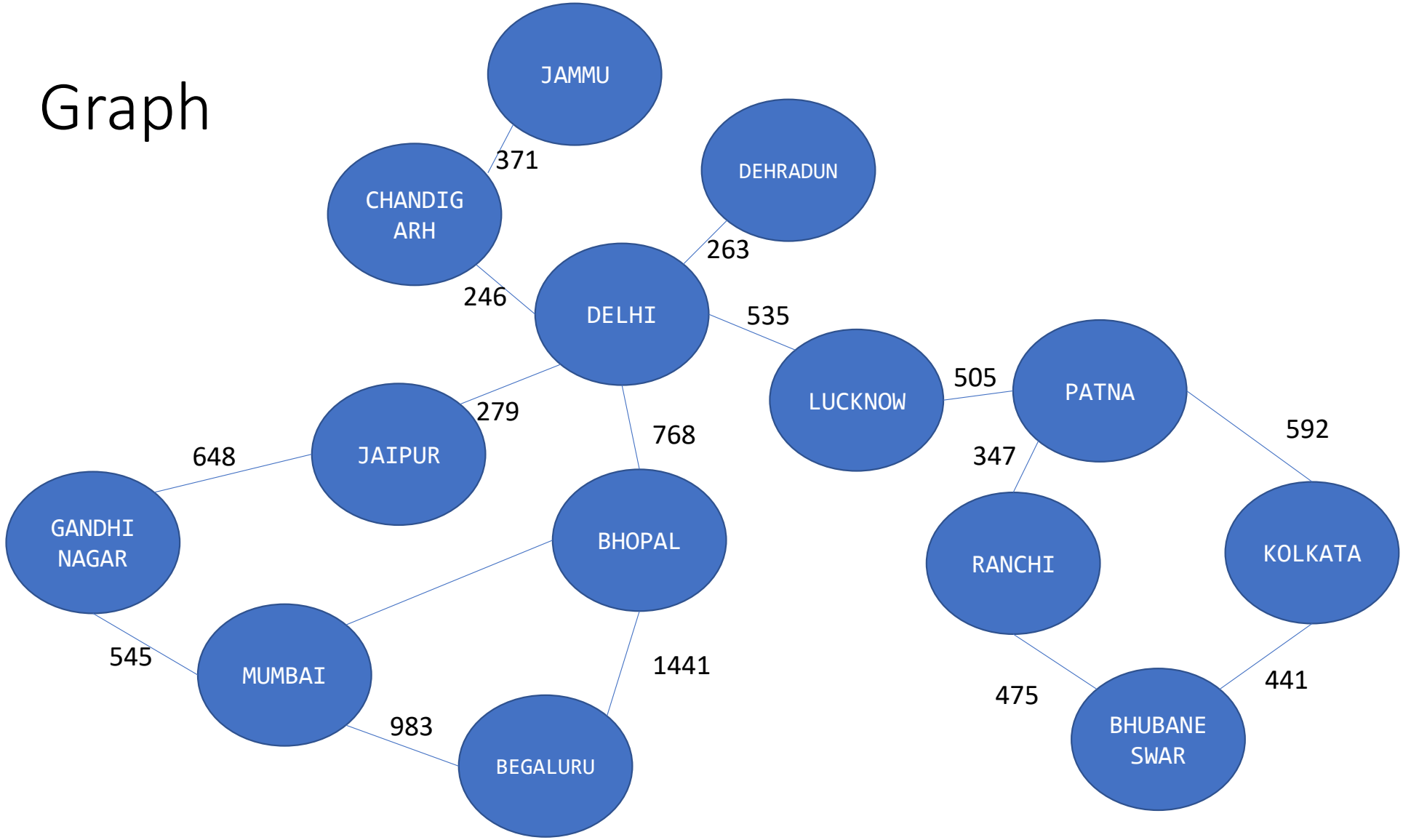
- The root node has no parent
- The other nodes have exactly one parent
- A node in the tree can have zero or more children

- Tree ADT:
  - root
  - insert
  - delete
  - children
  - parent

Example

- How to store the list of cities and the distance between neighboring cities?

Graph


JAMMU
371
CHANDIG ARH
DEHRADUN
263
246
DELHI
535
LUCKNOW
505
PATNA
279
JAIPUR
768
592
648
347
GANDHI NAGAR
BHOPAL
RANCHI
KOLKATA
545
MUMBAI
1441
475
441
983
BHUBANE SWAR
BEGALURU

Graph

- A set of vertices and edges
- Edges may have weights

- Graph ADT:
  - vertices
  - edges
  - addVertex
  - removeVertex
  - addEdge
  - removeEdge
  - incomingEdges
  - outgoingEdges

Revision of recursion

Reference

- Read section-3.5 from Goodrich, Tamassia, Mount book

Recursion

- A function calling itself is called recursion

$5! = 5 \times 4 \times 3 \times 2 \times 1$

Factorial:

0! = 1

n! = n * (n-1) * (n-2) * … * 3 * 2 *1

Factorial

- Recursive definition of factorial

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & n > 0 \end{cases}$$

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

Recursion

- Divide and conquer approach
  - **Divide** the problem into sub-problems of a similar type
  - **Conquer** the subproblem by solving them recursively
  - **Combine** the results of the subproblems to compute the result

- Structure of a recursive solution
  - One or two **base cases**
    - returns a value without making a recursive call
    - e.g., n=0 case in the factorial solution
  - One or two **recursive steps**
    - make recursive calls corresponding to the sub-problems
    - e.g., n*(n-1)! step in the factorial solution

Factorial

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

int

DEPTH	FN	RETURN VALUE
~~0~~	~~5!~~	~~r = 5 x factorial(4)~~
~~1~~	~~4!~~	~~r = 4 x factorial(3)~~
~~2~~	~~3!~~	~~r = 3 x factorial(2)~~
~~3~~	~~2!~~	~~r = 2 x factorial(1)~~
~~4~~	~~1!~~	~~r = 1 x factorial(0)~~
~~5~~	~~0!~~	~~r = 1~~
4	1!	r = 1 x 1
3	2!	r = 2 x 1
2	3!	r = 3 x 2 = 6
1	4!	r = 4 x 6 = 24
0	5!	r = 5 x 24 = 120

Factorial

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

DEPTH	FN	RETURN VALUE
0	5!	r = 5 * factorial(4)
1	4!	r = 4 * factorial(3)
2	3!	r = 3 * factorial(2)
3	2!	r = 2 * factorial(1)
4	1!	r = 1 * factorial(0)
5	0!	1
4	1!	1 * 1
3	2!	2 * 1
2	3!	3 * 2
1	4!	4 * 6
0	5!	5 * 24

x^n

$$\begin{cases} 1 & n = 0 \\ x * x^{n-1} & n > 0 \end{cases}$$

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```

power(x, n)

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```

DEPTH	FN	RETURN VALUE
~~0~~	~~3^5~~	~~r = 3 * power(3, 4)~~
~~1~~	~~3^4~~	~~r = 3 * power(3, 3)~~
~~2~~	~~3^3~~	~~r = 3 * power(3, 2)~~
~~3~~	~~3^2~~	~~r = 3 * power(3, 1)~~
~~4~~	~~3^1~~	~~r = 3 * power(3, 0)~~
~~5~~	~~3^0~~	~~1~~
4	3^1	3 * 1 = 3
3	3^2	3 * 3 = 9
2	3^3	3 * 9 = 27
1	3^4	3 * 27
0	3^5	3 * 81

power(x, n)

- Can we reduce computation?

power(x, n)

- Properties of power

$$x^{100} = x^{50} * x^{50}$$
$$x^{101} = x^{50} * x^{50} * x$$

$$x^n$$

$$\begin{cases} 1 & if\ n = 0 \\ x^{\frac{n}{2}} * x^{\frac{n}{2}} & if\ n\ \%\ 2 = 0 \\ x * x^{\frac{n-1}{2}} * x^{\frac{n-1}{2}} & if\ n\ \%\ 2 = 1 \end{cases}$$

power_f(x, n)

```
int power_f(int x, int n) {
    int pow_h;
    if (n == 0)
        return 1;
    if ((n % 2) == 0) {
        pow_h = power_f(x, n/2);
        return pow_h * pow_h;
    }
    else {
        pow_h = power_f(x, (n-1)/2);
        return x * pow_h * pow_h;
    }
}
```

DEPTH	FN	RETURN VALUE
0	3^5	t=power_f (3, 2); r= 3×t×t
~~1~~	~~3^2~~	~~t = power_f (3, 1), r = t×t~~
~~2~~	~~3^1~~	~~t = power_f(3, 0) r=3×t×t~~
~~3~~	~~3^0~~	~~r~~ = 1
2	3^1	t = 1 ; r = 3
1	3^2	t = 3 ; r = 3×3 = 9
0	3^5	t = 9 ; r = 3×9×9

power_f(x, n)

```
int power_f(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power_f(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power_f(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```

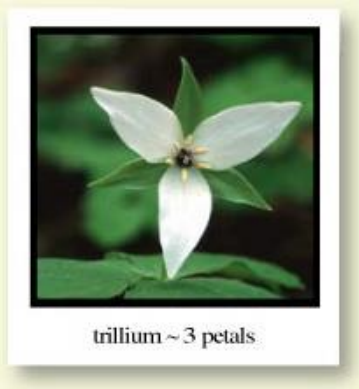
DEPTH	FN	RETURN VALUE
0	3^5	r = power_f(3, 2); 3*r*r
1	3^2	r = power_f(3, 1); r*r
2	3^1	r = power_f(3, 0); 3*r*r
3	3^0	1
2	3^1	3 * 1 * 1
1	3^2	3 * 3
0	3^5	3 * 9 * 9

Fibonacci numbers

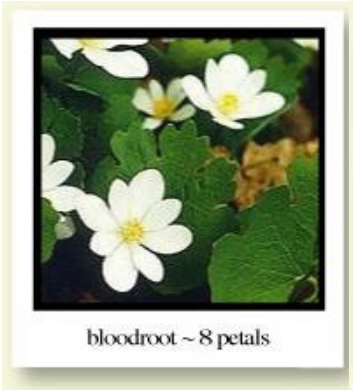
- Fibonacci numbers are a sequence of numbers in which each number is the sum of the two preceding numbers

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Fibonacci numbers in nature


trillium ~ 3 petals

wild rose ~ 5 petals


bloodroot ~ 8 petals

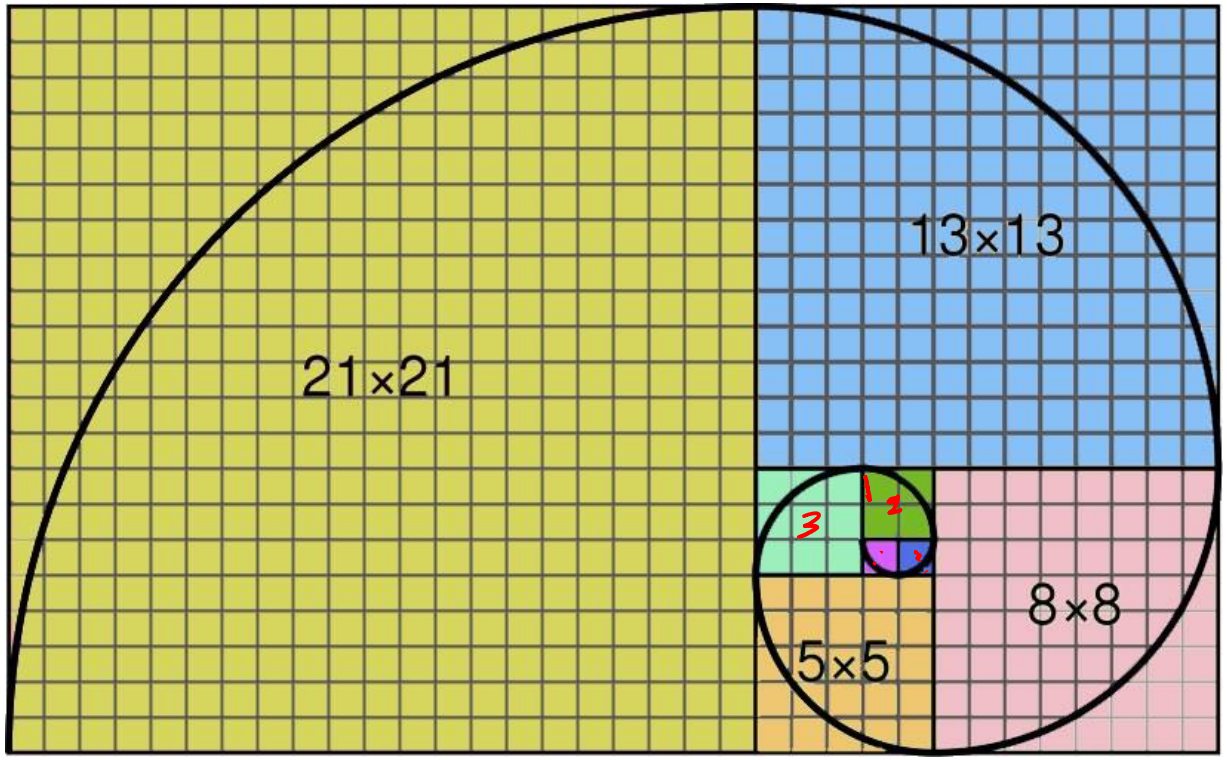
black eyed susan ~ 13 petals

chamomile ~ 21 petals

gaillardia ~ 34 petals

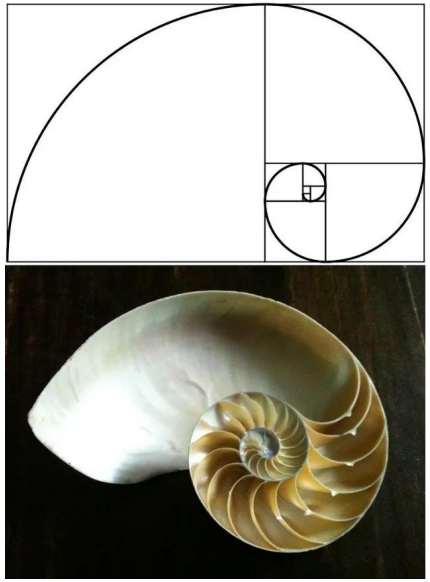
1 1 2 3 5 8 13 21

Fibonacci spiral



Fibonacci spiral

Galaxy



Sea shell

Golden ratio

2 / 1	2
3 / 2	1.5
5 / 3	1.666
8 / 5	1.6
13 / 8	1.625
21 / 13	1.615
34 / 21	1.619
55 / 34	1.617
89 / 55	1.618
144 / 89	1.617
233 / 144	1.618
377 / 233	1.618

Golden ratio

- The golden ratio (ϕ) is 1.618033988749 …
- Many patterns based on the golden ratio exist in the nature

Golden ratio

Ancient temple in Greece almost fits into a golden rectangle.
We don’t know for sure if the temple was designed that way.

Fibonacci numbers

- Recursive definition of Fibonacci numbers

$$\mathrm{f}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

$f(2) = f(1) + f(0) = 1$
$f(3) = f(2) + f(1) = 2$
$f(4) = f(3) + f(2) = 3$
...

Fibonacci numbers

- Recursive definition of Fibonacci numbers

$$\mathrm{f}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

Fibonacci numbers

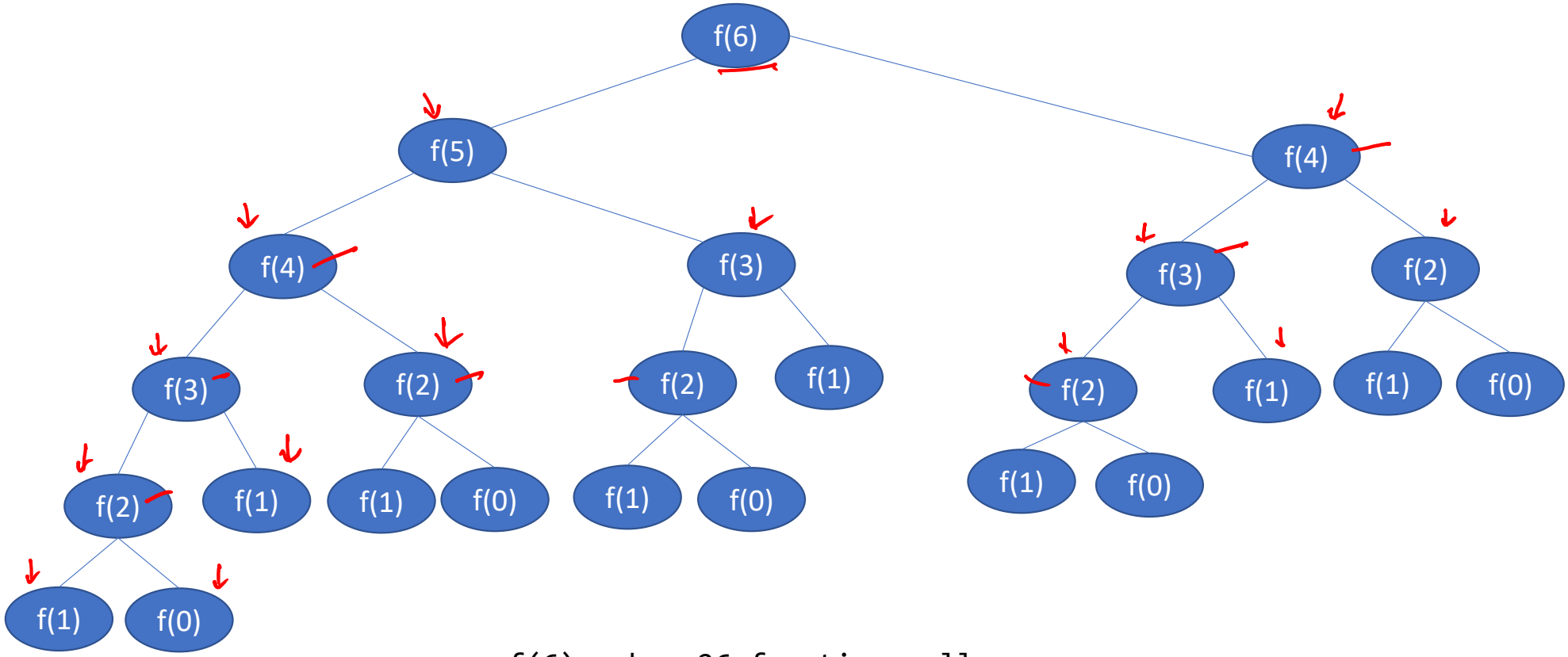
```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

DEPTH	FN	RETURN VALUE
	fib(4)	

Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

DEPTH	FN	RETURN VALUE
0	fib(4)	r = fib(3) + fib(2)
1	fib(3)	r = fib(2) + fib(1)
2	fib(2)	r = fib(1) + fib(0)
3	fib(1)	r = 1
2	fib(2)	r = 1 + fib(0)
3	fib(0)	r = 0
2	fib(2)	r = 1 + 0 = 1
1	fib(3)	r = 1 + fib(1)
2	fib(1)	r = 1
1	fib(3)	r = 1 + 1 = 2
0	fib(4)	r = 2 + fib(2)
1	fib(2)	r = fib(1) + fib(0)
2	fib(1)	r = 1
1	fib(2)	r = 1 + fib(0)
2	fib(0)	r = 0
1	fib(2)	r = 1 + 0 = 1
0	fib(4)	r = 2 + 1 = 3



f(6) makes 26 function calls.

Faster recursive solution

$f(n-1) + f(n-2)$

$f(n-1)$

$[f(n-1), f(n-2)]$

- Base case
  - `if (n == 1) return (1, 0)`

- Recursive step
  - Recursively call for `n-1` to obtain `(x, y) = (f(n-1), f(n-2))`
  - `return (x+y, x)`

fib(2

Fibonacci numbers

```
void fib(int n, int f_n[1], int f_n_minus_1[1])
{
   if (n == 1) {
      f_n[0] = 1;
      f_n_minus_1[0] = 0;
      return;
   }
   fib(n-1, f_n, f_n_minus_1);
   int newval = f_n[0] + f_n_minus_1[0];
   f_n_minus_1[0] = f_n[0];
   f_n[0] = newval;
}
```

f_n f(1)
f_n-1 f(0)

DEPTH	FN	RETURN VALUE
0	fib(6)	fib(5, f_n, f_n_minus_1) f_n f_n_minus
~~1~~	~~fib(5)~~	~~fib(4, f_n, f_n_minus_1)~~ f_n f_n_minus
~~2~~	~~fib(4)~~	~~fib(3, . .)~~ f_n f_n-1
~~3~~	~~fib(3)~~	~~fib(2, . .)~~ f_n f_n-1
~~4~~	~~fib(2)~~	~~fib(1, . .)~~ f_n f_n-1
~~5~~	~~fib(1)~~	f_n = 1 f_n-1 = 0
4	fib(2)	1, 0 f_n = 1 f_n-1 = 1
3	fib(3)	1, 1 f_n = 2 f_n-1 = 1
2	fib(4)	2, 1 f_n = 3 f_n-1 = 2
1	fib(5)	3, 2 f_n = 5 f_n-1 = 3
0	Fib(6)	5, 3 f_n = 8, f_n-1 = 5

Fibonacci numbers

main() { fib(5, 0, 0);
}

fib(int n, int f_n, int f_n_minus_1) {
if (n == 1) {
f_n = 1
f_n_minus_1 = 0
return;
}

```
void fib(int n, int f_n[1], int f_n_minus_1[1])
{
   if (n == 1) {
      f_n[0] = 1;
      f_n_minus_1[0] = 0;
      return;
   }
   fib(n-1, f_n, f_n_minus_1);
   int newval = f_n[0] + f_n_minus_1[0];
   f_n_minus_1[0] = f_n[0];
   f_n[0] = newval;
}
```

DEPTH	FN	RETURN VALUE
0	fib(6)	fib(5, f_n, f_n_minus_1)
1	fib(5)	fib(4, f_n, f_n_minus_1)
2	fib(4)	fib(3, f_n, f_n_minus_1)
3	fib(3)	fib(2, f_n, f_n_minus_1)
4	fib(2)	fib(1, f_n, f_minus_1)
5	fib(1)	f_n = 1, f_minus_1 = 0
4	fib(2)	f_n = 1, f_n_minus_1 = 1
3	fib(3)	f_n = 2, f_minus_1 = 1
2	fib(4)	f_n = 3, f_n_minus_1 = 2
1	fib(5)	f_n = 5, f_n_minus_1 = 3
0	fib(6)	f_n = 8, f_n_minus_1 = 5

Linear search

- Let `arr` be an input array of length `n`. Given a value `x`, we want to find an index i (`0 <= i < n`), such that `arr[i] == x`. If no such index exists, then the algorithm returns `-1`.

Linear search

12	11	23	13	41	19	25
0	1	2	3			

```
int lsearch(int arr[], int len, int val);
val = 30;
val = 13;
```

Iterative solution

i = i-1

```
int lsearch(int arr[], int val, int len) {
   int i;
   for (i = len-1; i >= 0; i--) {
      if (arr[i] == val)
         return i;
   }
   return -1;
}
```

Recursive solution

- `int lsearch(int arr[], int val, int len);`
  - Initially, `len` is the length of the input `arr`
  - `val` is the value being searched
- Base cases
  - `if (len == 0) return -1`
  - `if (arr[len-1] == val) return len-1`
- Recursive step
  - Decrement the length of the array and recursively call search

Recursive algorithm

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

Binary search

- Let `arr` be an input sorted array (in ascending order) of length `n`. Given a value x, we want to find an index i (`0 <= i < n`), such that `arr[i] == x`. If no such index exists, then the algorithm returns `-1`.

Binary search

$\boxed{0}$ 1 mid $\frac{1+1}{2} = 1$

mid $= \frac{lo+hi}{2} = 5$

mid $= \frac{0+4}{2} = 2$

mid $= \frac{0+1}{2} = 0$

10, hi

6, 10

6, 7

6

7, 7

7, 7

11	15	19	23	41	59	65	75	79	81	87
lo= 0	1	2	3	4	5	6	7	8	9	10 = hi

```
int bsearch(int arr[], int len, int val);
val = 15;
val = 74;
```

Recursive solution

- `int bsearch(int arr[], int val, int lo, int hi);`
  - Initially, `lo` = 0 and hi = length(`arr) - 1`
  - `val` is the value being searched
- Base cases
  - `if (lo > hi) return -1`
  - `mid = (lo + hi)/2`
  - `if (arr[mid] == val) return mid`
- Recursive step
  - If `val` > `arr[mid],` search the second half (`mid+1, lo`), recursively
  - Otherwise, search the first half (`lo, mid-1`)

Recursive solution

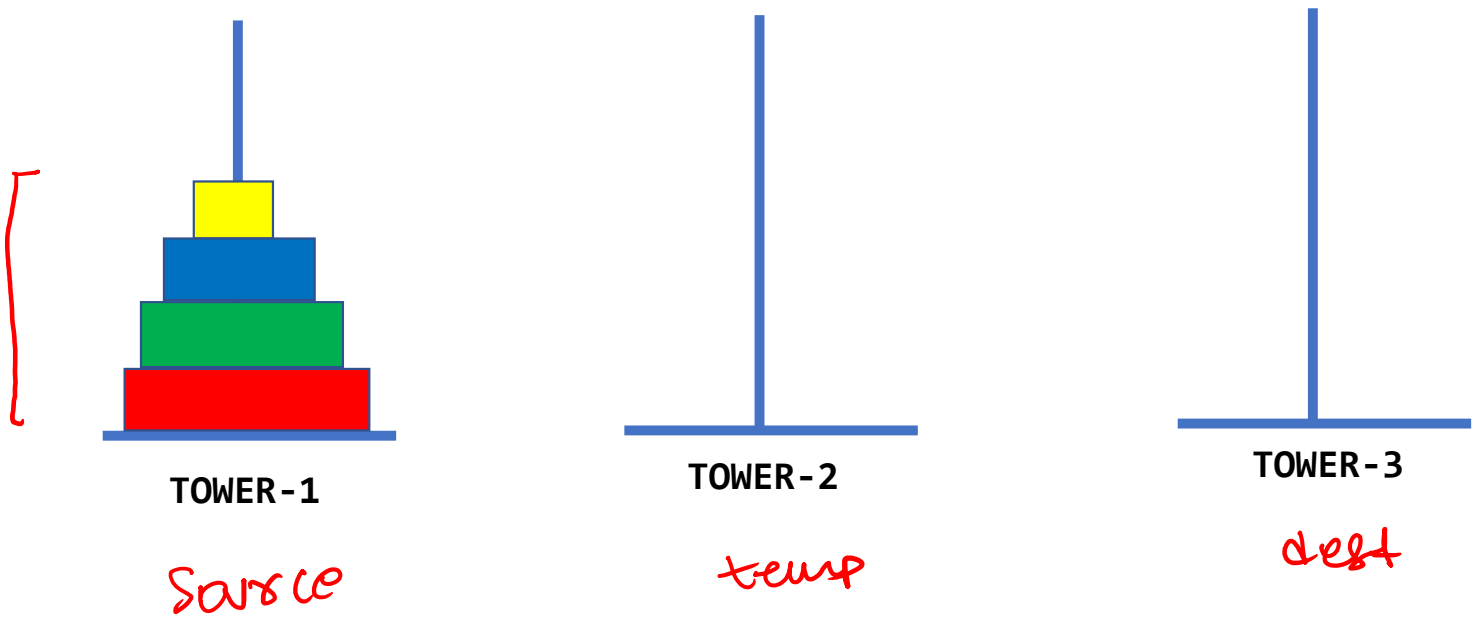
```
int bsearch(int arr[], int val, int lo, int hi) {
   if (hi < lo)
      return -1;
   int mid = (lo + hi) / 2;
   if (arr[mid] == val)
      return mid;
   if (arr[mid] > val)
      return bsearch(arr, val, lo, mid-1);
   else
      return bsearch(arr, val, mid+1, hi);
}
```

7, 7

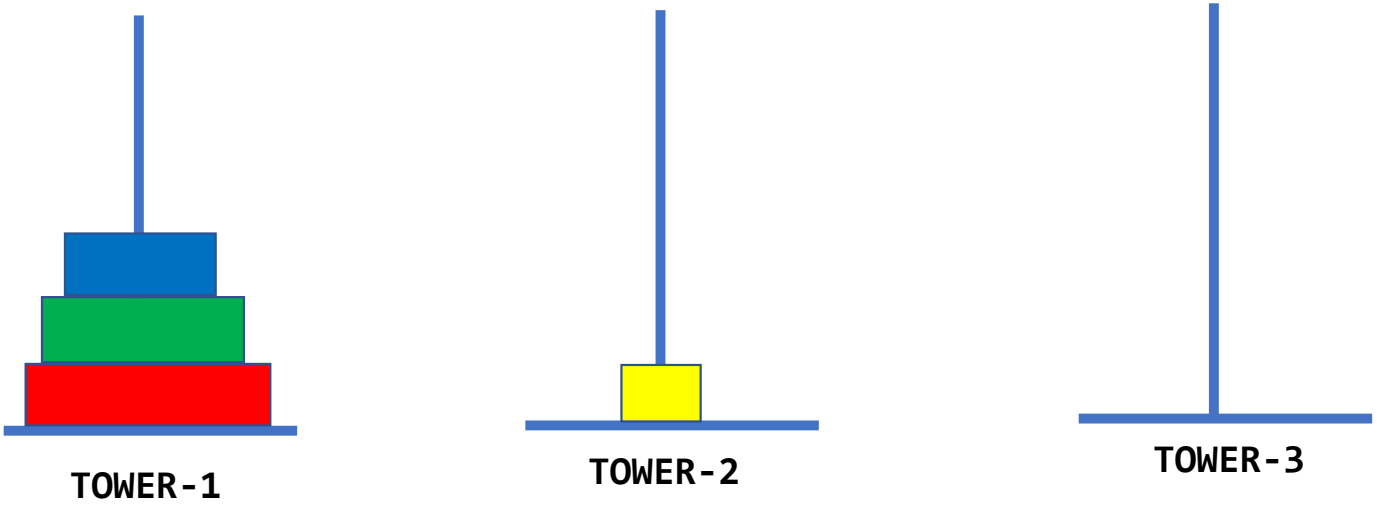
Tower of Hanoi

- Three towers: source, middle (or temporary), destination
- n discs of different sizes are stacked on the source tower
- The discs are ordered from largest (bottom) to smallest (top)
- The goal is to move all discs from source to destination
- Rules for movement
  - You can move only one disc at a time from one tower to another
  - A larger disc can’t be placed over a smaller disc

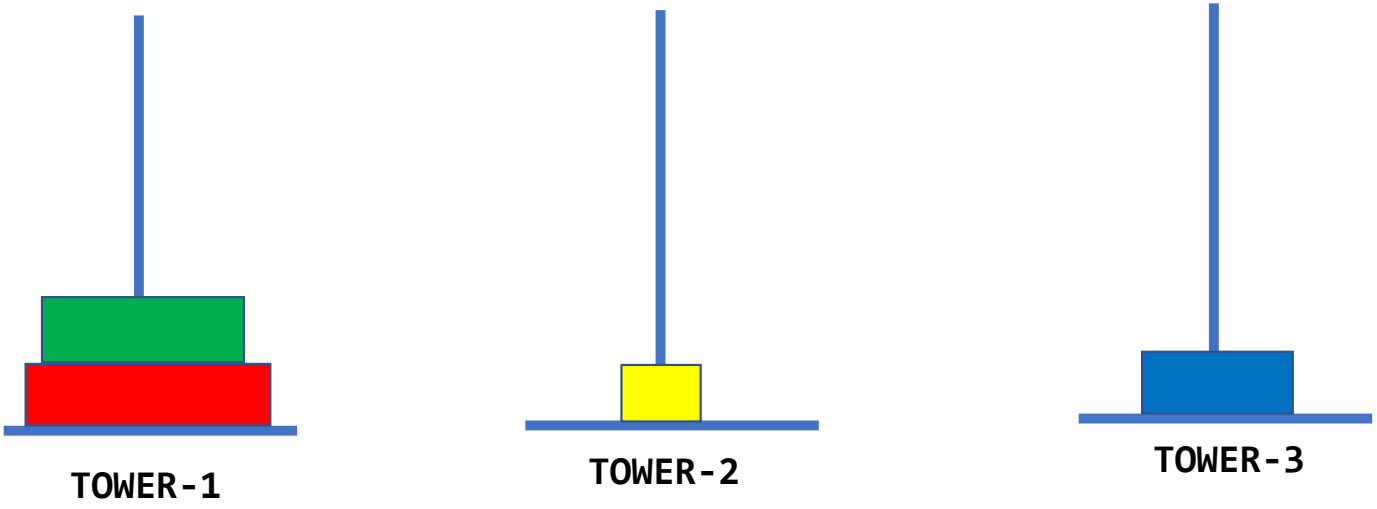
Tower of Hanoi



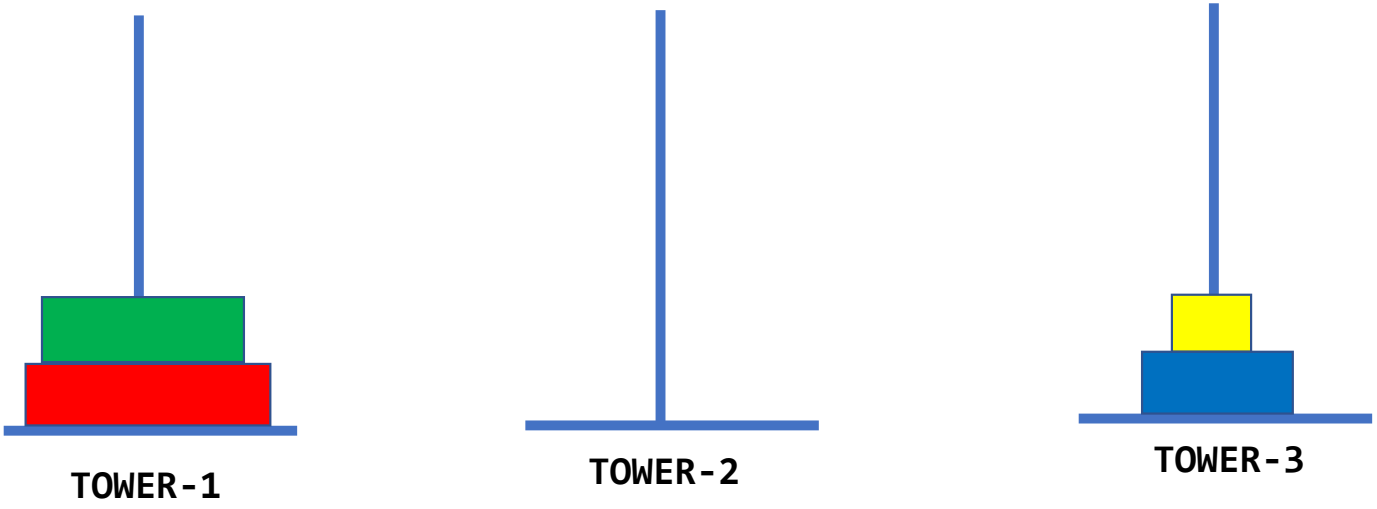
Tower of Hanoi



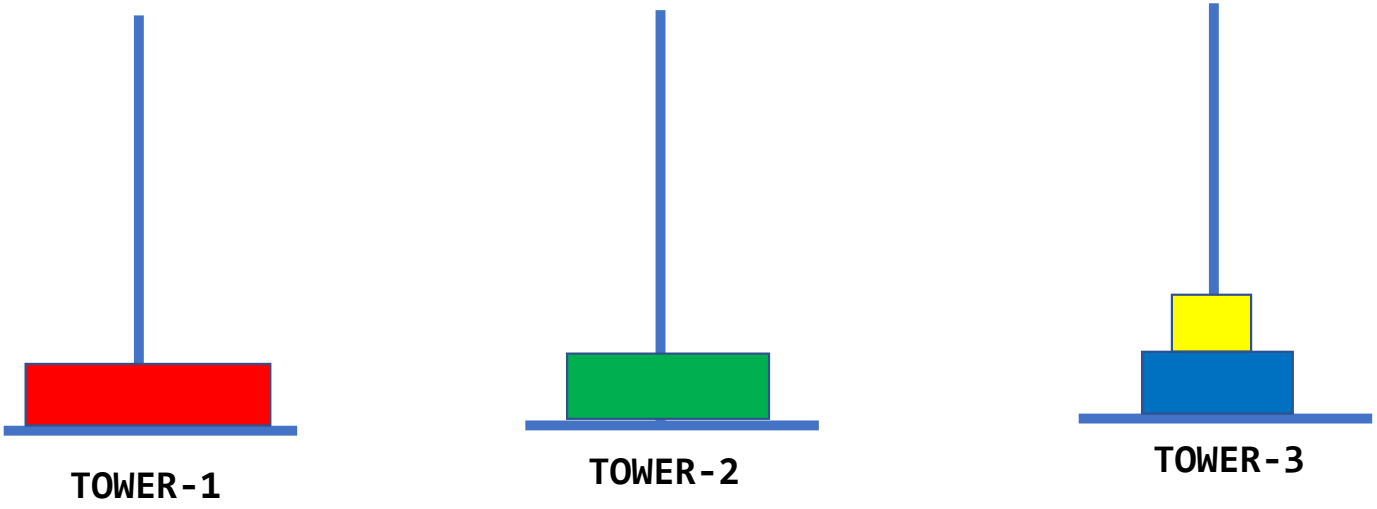
Tower of Hanoi



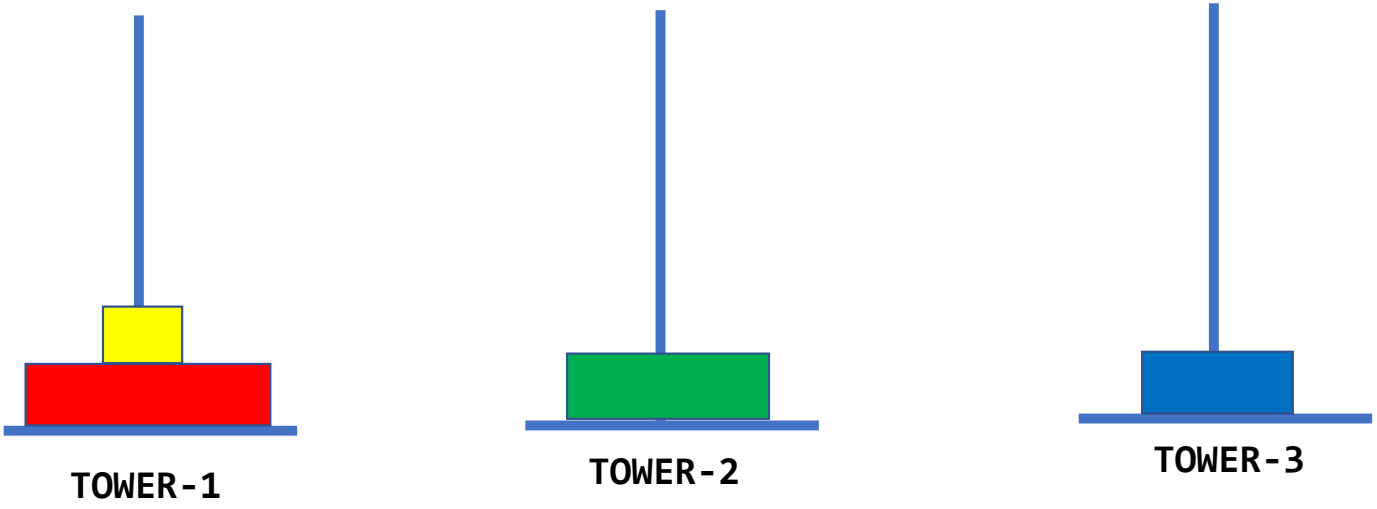
Tower of Hanoi



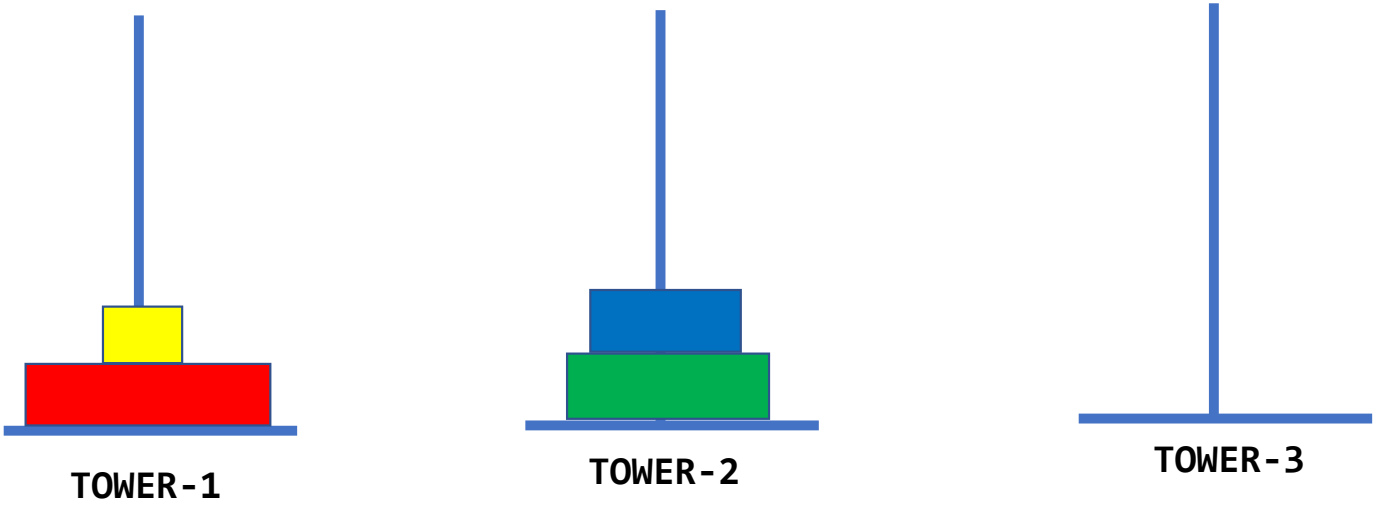
Tower of Hanoi



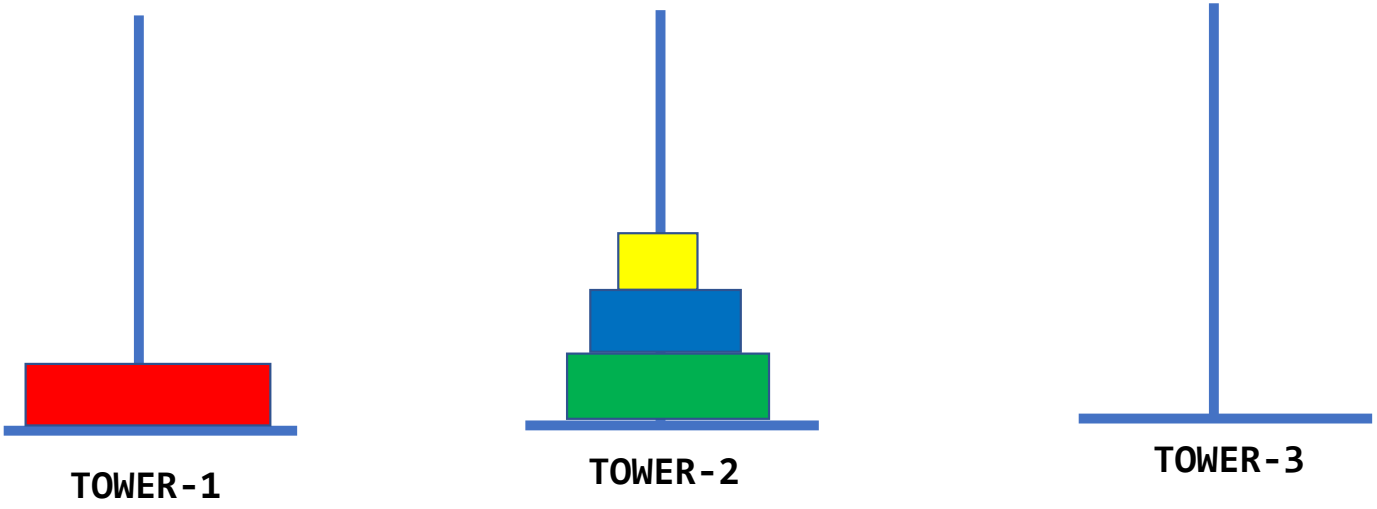
Tower of Hanoi



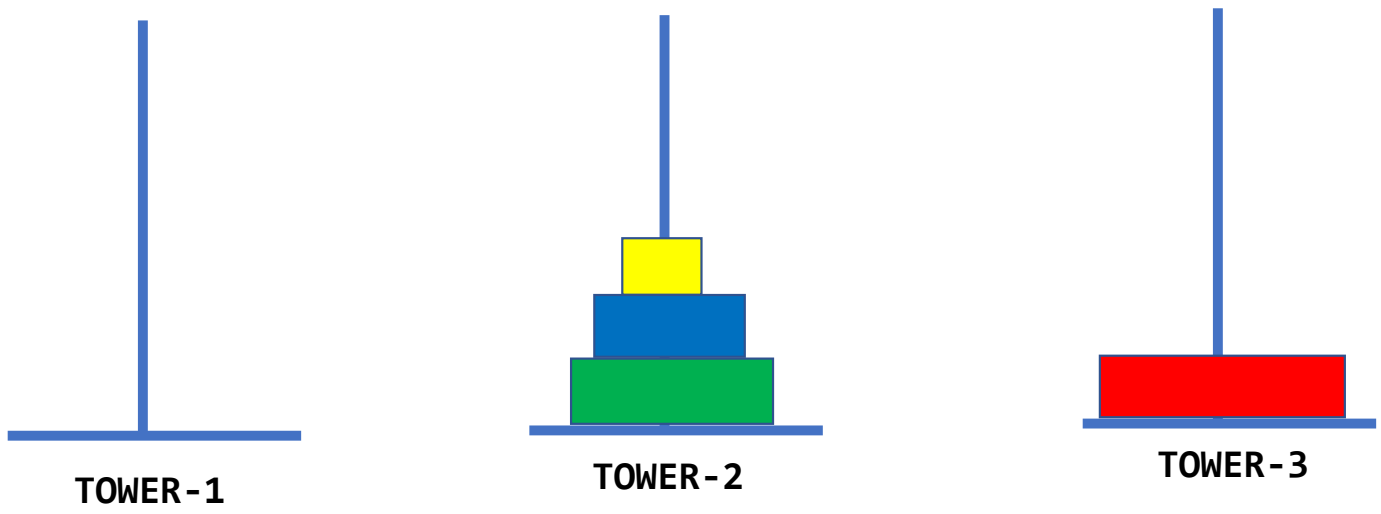
Tower of Hanoi



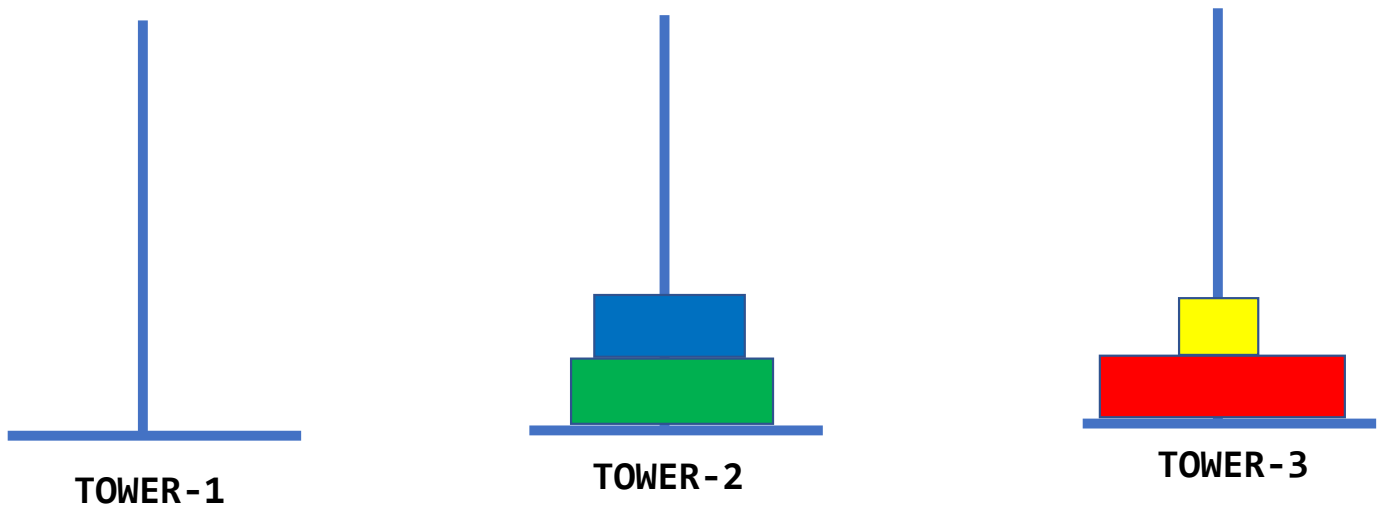
Tower of Hanoi



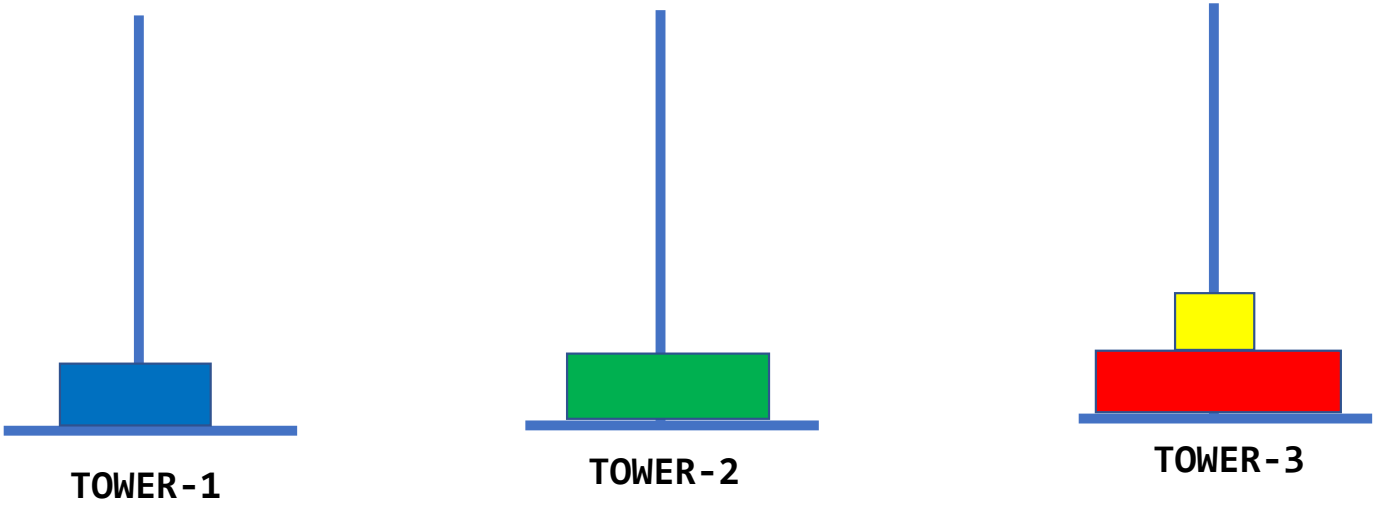
Tower of Hanoi



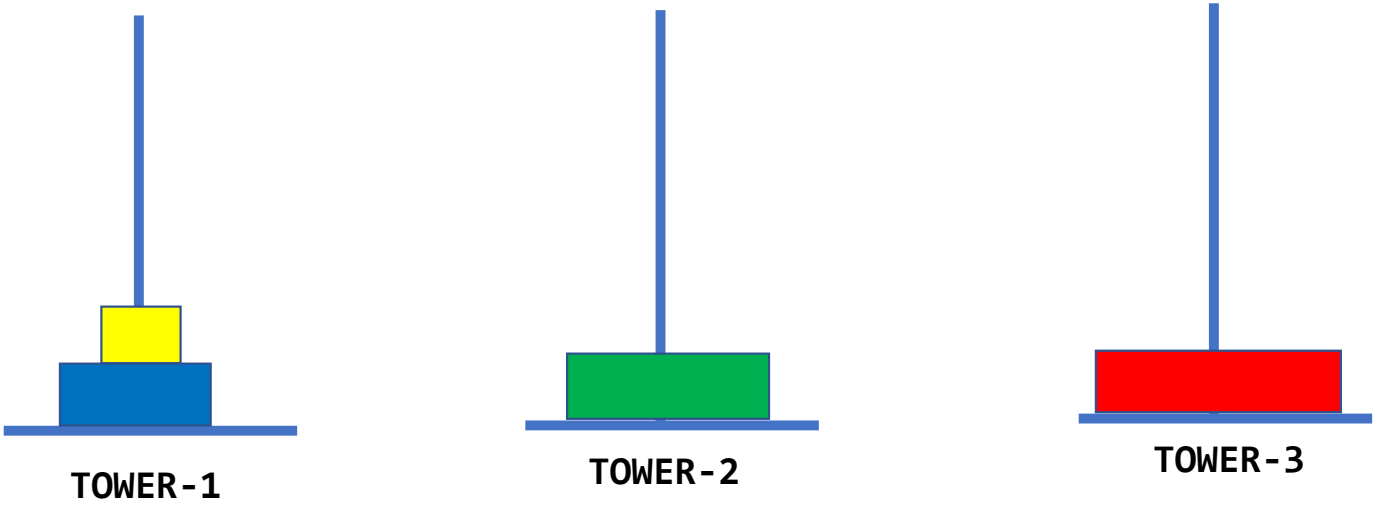
Tower of Hanoi



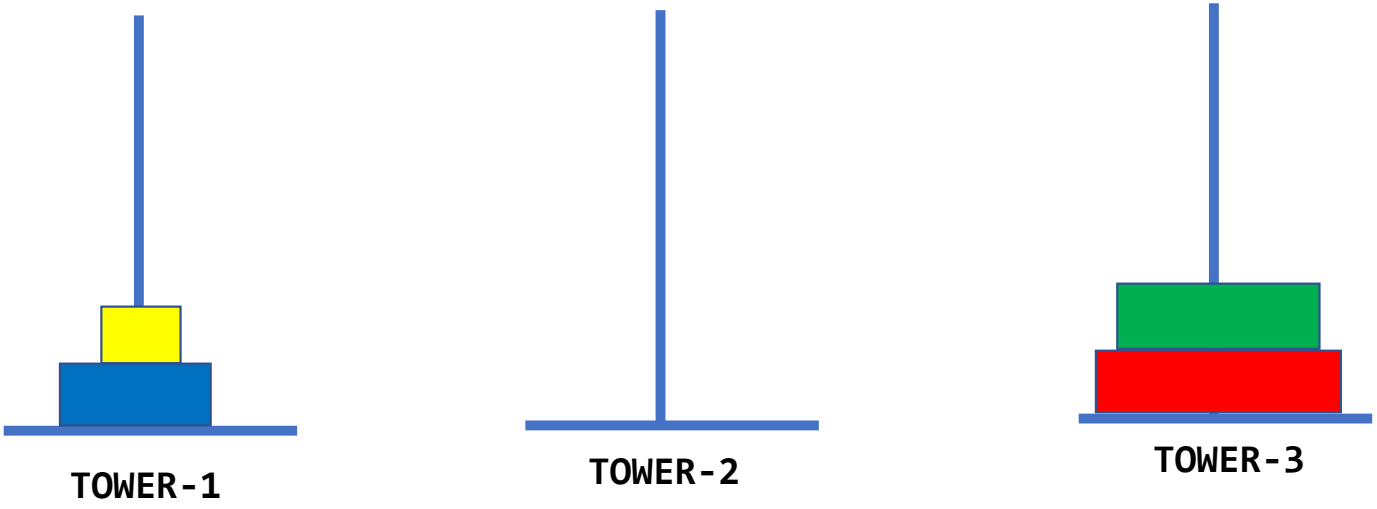
Tower of Hanoi



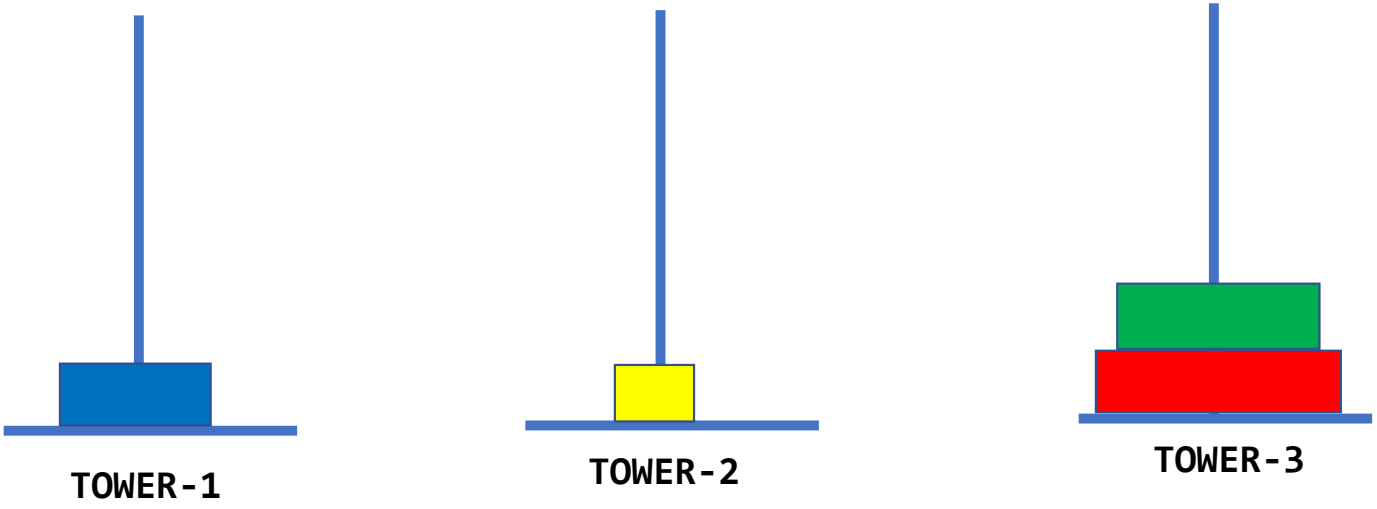
Tower of Hanoi



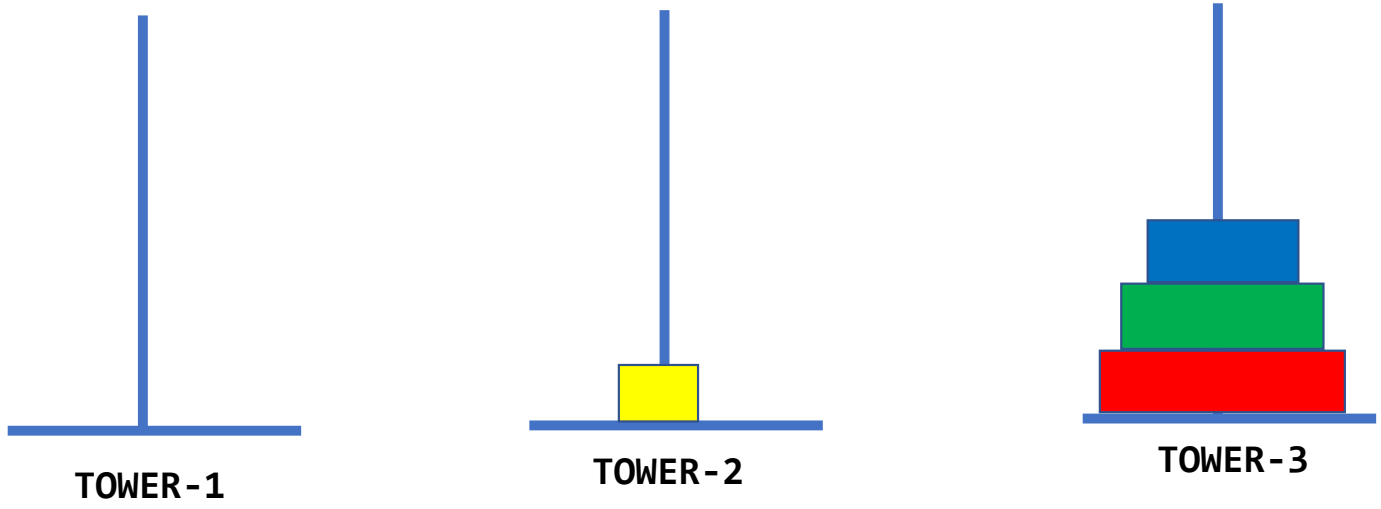
Tower of Hanoi



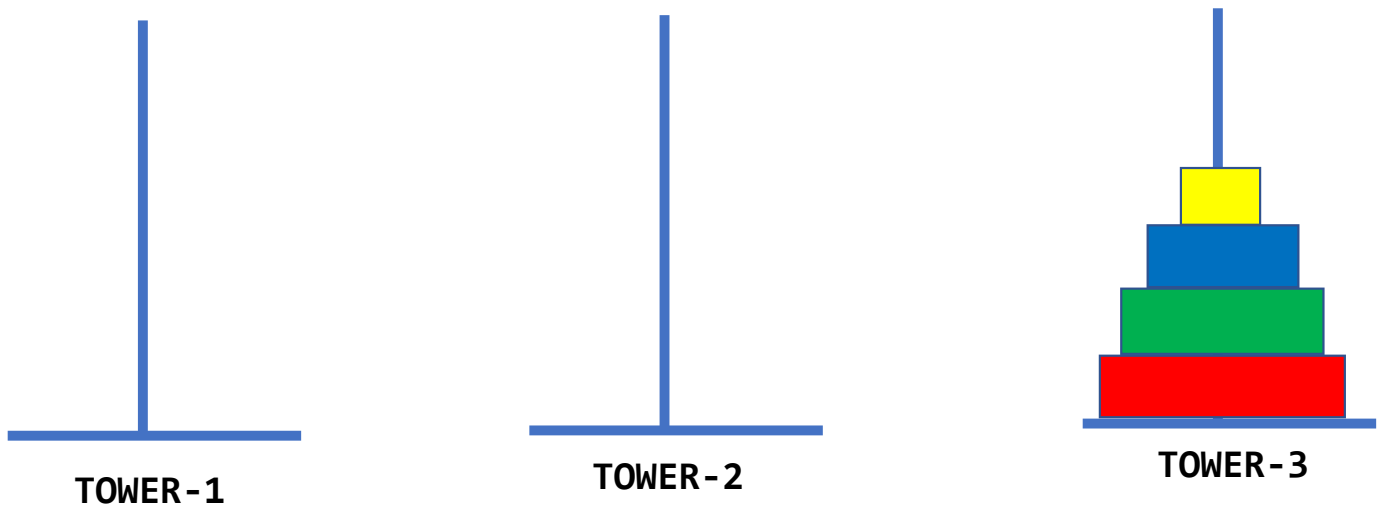
Tower of Hanoi



Tower of Hanoi

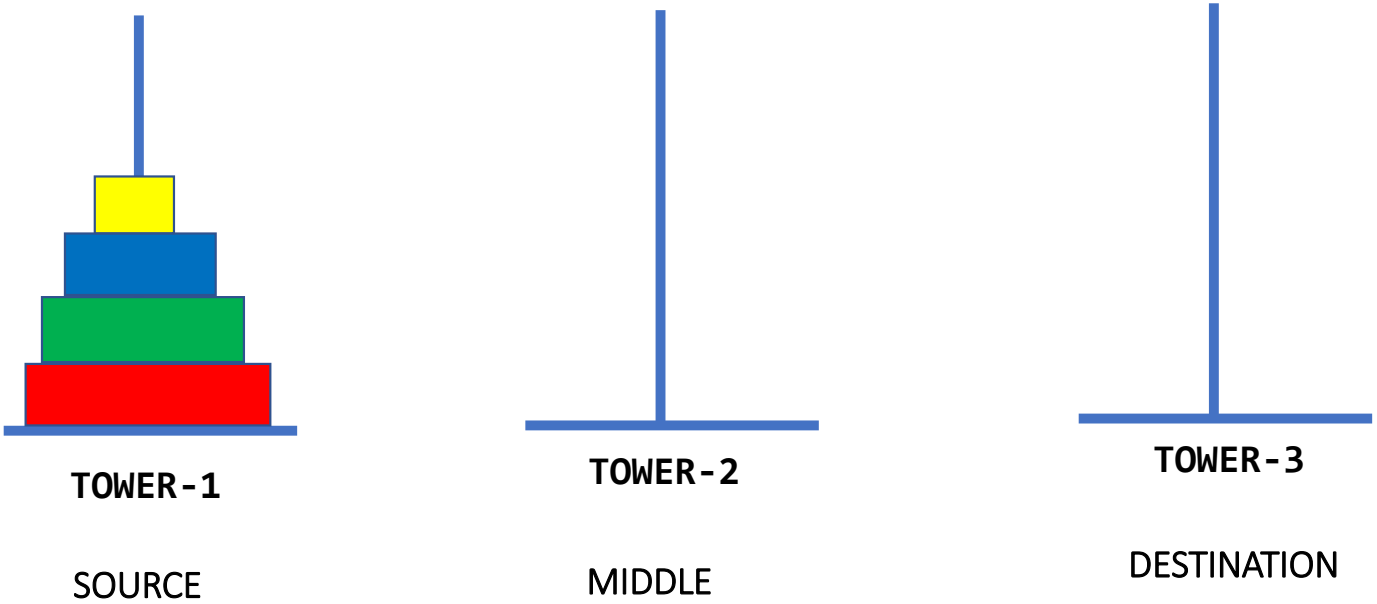


Tower of Hanoi



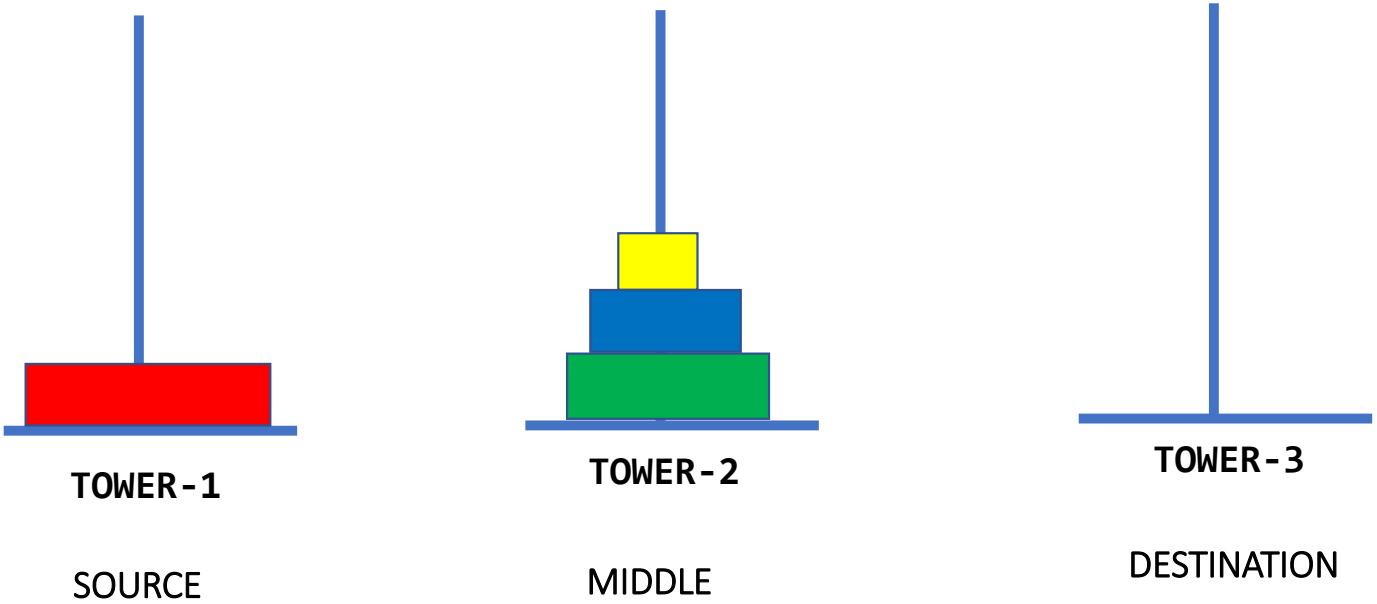
Initial state

- Goal: move n discs from source to destination

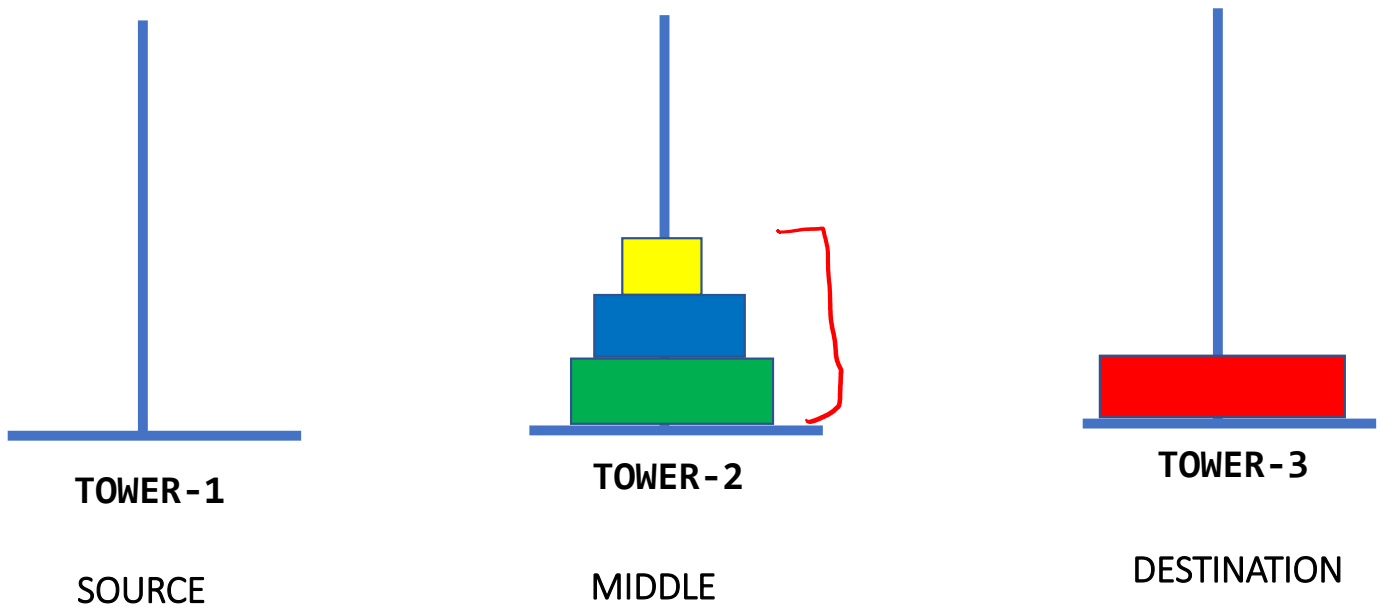


Intermediate state

- After moving n-1 discs from the source tower to the middle tower using the destination tower as temporary

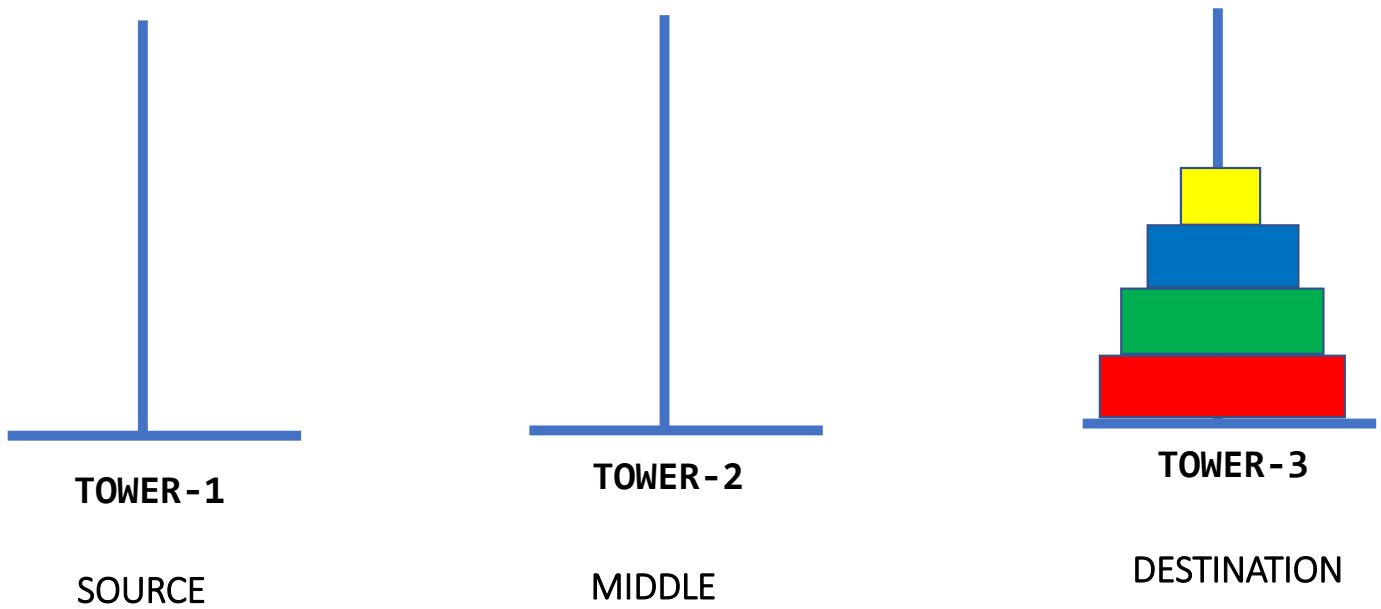


Move bottom disc from source to destination



Final state

- After moving n-1 discs from the middle tower to the destination tower using the source tower as temporary



Tower of Hanoi

```
void move(int n, char src_t[], char dst_t[], char mid_t[]) {
  if (n == 0)
    return;
  move(n-1, src_t, mid_t, dst_t);
  printf("moving from %s to %s\n", src_t, dst_t);
  move(n-1, mid_t, dst_t, src_t);
}

int main() {
  move(4, "Tower1", "Tower3", "Tower2");
  return 0;
}
```

Today’s lecture

- Recursion
- Efficiency of algorithms

Factorial

- Recursive definition of factorial

$$n! = \begin{array}{cc} 1 & n = 0 \\ n * (n-1)! & n > 0 \end{array}$$

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

x^n

$$\begin{cases} 1 & n = 0 \\ x * x^{n-1} & n > 0 \end{cases}$$

```
int power(int x, int n) {
   if (n == 0)
      return 1;
   return x * power(x, n-1);
}
```

Faster x^n

```
int power_f(int x, int n) {
   int pow_h;
   if (n == 0)
      return 1;
   if ((n % 2) == 0) {
      pow_h = power_f(x, n/2);
      return pow_h * pow_h;
   }
   else {
      pow_h = power_f(x, (n-1)/2);
      return x * pow_h * pow_h;
   }
}
```

DEPTH	FN	RETURN VALUE
0	3^5	r = power_f(3, 2); 3*r*r
1	3^2	r = power_f(3, 1); r*r
2	3^1	r = power_f(3, 0); 3*r*r
3	3^0	1
2	3^1	3 * 1 * 1
1	3^2	3 * 3
0	3^5	3 * 9 * 9

Fibonacci numbers

- Recursive definition of Fibonacci numbers

$$\mathrm{f}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

Faster Fibonacci numbers

```
void fib(int n, int f_n[1], int f_n_minus_1[1])
{
   if (n == 1) {
      f_n[0] = 1;
      f_n_minus_1[0] = 0;
      return;
   }
   fib(n-1, f_n, f_n_minus_1);
   int newval = f_n[0] + f_n_minus_1[0];
   f_n_minus_1[0] = f_n[0];
   f_n[0] = newval;
}
```

Binary search

```
int bsearch(int arr[], int val, int lo, int hi) {
   if (hi < lo)
      return -1;
   int mid = (lo + hi) / 2;
   if (arr[mid] == val)
      return mid;
   if (arr[mid] > val)
      return bsearch(arr, val, lo, mid-1);
   else
      return bsearch(arr, val, mid+1, hi);
}
```

Tower of Hanoi

```
void move(int n, char src_t[], char dst_t[], char mid_t[]) {
  if (n == 0)
    return;
  move(n-1, src_t, mid_t, dst_t);
  printf("moving from %s to %s\n", src_t, dst_t);
  move(n-1, mid_t, dst_t, src_t);
}

int main() {
  move(4, "Tower1", "Tower3", "Tower2");
  return 0;
}
```

Selection sort

- Find the minimum element and swap it with the first element
- Find the second minimum and swap it with the second element
- Find the third minimum and swap it with the third element

- and so on.

- Find the $(n-1)th$ minimum and swap it with the $(n-1)th$ element

Selection sort

83	86	77	15	93	35	86	92	49	21

Selection sort

83	86	77	15	93	35	86	92	49	21

15	86	77	83	93	35	86	92	49	21

Selection sort

83	86	77	15	93	35	86	92	49	21

15	86	77	83	93	35	86	92	49	21

15	21	77	83	93	35	86	92	49	86

Selection sort

83	86	77	15	93	35	86	92	49	21

15	86	77	83	93	35	86	92	49	21

15	21	77	83	93	35	86	92	49	86

15	21	35	83	93	77	86	92	49	86

Selection sort

15	21	35	83	93	77	86	92	49	86

15	21	35	49	93	77	86	92	83	86

15	21	35	49	77	93	86	92	83	86

15	21	35	49	77	83	86	92	93	86

Selection sort

15	21	35	49	77	83	86	92	93	86

15	21	35	49	77	83	86	92	93	86

15	21	35	49	77	83	86	86	93	92

15	21	35	49	77	83	86	86	92	93

Selection sort

```
void selection_sort(int arr[], int n)
{
  int i, idx;

  for (i = 0; i < n; i++) {
    idx = find_min(arr, i, n);
    if (idx != i) {
      swap(arr, i, idx);
    }
  }
}
```

```
int find_min(int arr[], int start, int end)
{
  int i, min, idx;

  min = arr[start];
  idx = start;
  for (i = start + 1; i < end; i++) {
    if (arr[i] < min) {
      min = arr[i];
      idx = i;
    }
  }
  return idx;
}

void swap(int arr[], int i, int j)
{
  int tmp = arr[i];
  arr[i] = arr[j];
  arr[j] = tmp;
}
```

Selection sort recursion

```
                                        0
void selection_sort_r(int arr[], int pos, int n)
{
  int idx;

  if (pos >= n) {
    return;
  }

  idx = find_min(arr, pos, n);
  if (idx != pos) {
    swap(arr, pos, idx);
  }
  selection_sort(arr, pos+1, n);
}
```

```
int find_min(int arr[], int start, int end)
{
  int i, min, idx;

  min = arr[start];
  idx = start;
  for (i = start + 1; i < end; i++) {
    if (arr[i] < min) {
      min = arr[i];
      idx = i;
    }
  }
  return idx;
}

void swap(int arr[], int i, int j)
{
  int tmp = arr[i];
  arr[i] = arr[j];
  arr[j] = tmp;
}
```

Tail recursion

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

```
arr: 13, 79, 87, 32, 65, 23, 11, 57, 23
search : 20
```

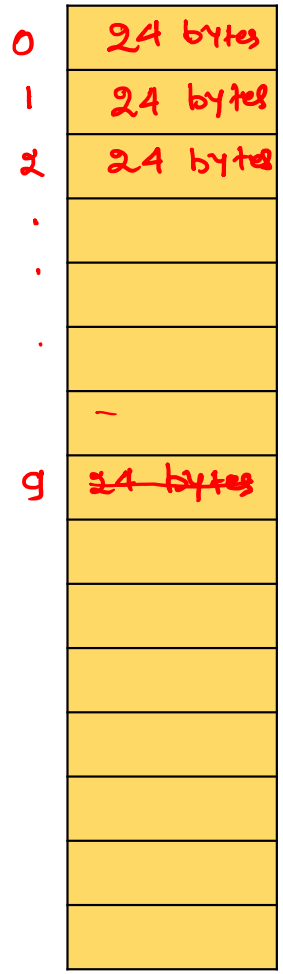
DEPTH	LEN	RETURN VALUE
0	9	lsearch(arr, 20, 8)
1	8	lsearch(arr, 20, 7)
2	7	lsearch(arr, 20, 6)
3	6	lsearch(arr, 20, 5)
4	5	lsearch(arr, 20, 4)
5	4	lsearch(arr, 20, 3)
6	3	lsearch(arr, 20, 2)
7	2	lsearch(arr, 20, 1)
8	1	lsearch(arr, 20, 0)
9	0	-1

Function arguments and local variables

- Space for function arguments and local variables are automatically allocated by the compiler
- The allocation strategy may differ across compilers
- Most compilers use a runtime stack to store a part of local variables, function arguments, and the caller info
- The caller info is required to return to the caller

Tail recursion

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

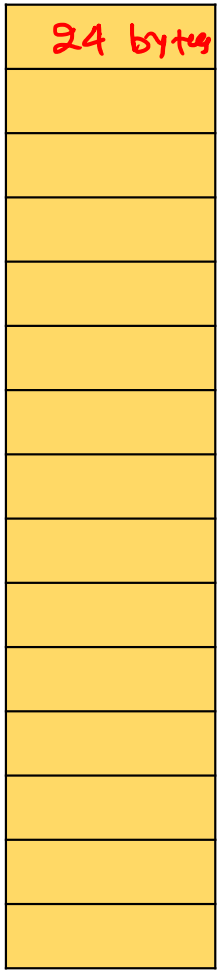


DEPTH	LEN	RETURN VALUE
0	9	lsearch(arr, 20, 8)
1	8	lsearch(arr, 20, 7)
2	7	lsearch(arr, 20, 6)
3	6	lsearch(arr, 20, 5)
4	5	lsearch(arr, 20, 4)
5	4	lsearch(arr, 20, 3)
6	3	lsearch(arr, 20, 2)
7	2	lsearch(arr, 20, 1)
8	1	lsearch(arr, 20, 0)
9	0	-1

```
arr: 13, 79, 87, 32, 65, 23, 11, 57, 23
search : 20
```

Linear search

```
int lsearch(int arr[], int val, int len) {
   int i;
   for (i = len-1; i >= 0; i--) {
      if (arr[i] == val)
         return i;
   }
   return -1;
}
```

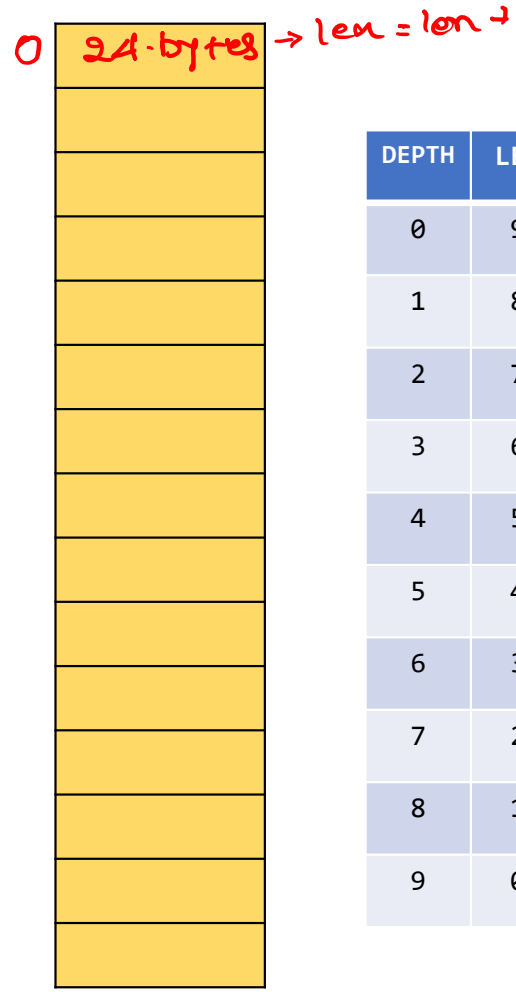


Tail recursion

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

The caller simply returns after the recursive call.

arr: 13, 79, 87, 32, 65, 23, 11, 57, 23
search : 20



DEPTH	LEN	RETURN VALUE
0	9	lsearch(arr, 20, 8)
1	8	lsearch(arr, 20, 7)
2	7	lsearch(arr, 20, 6)
3	6	lsearch(arr, 20, 5)
4	5	lsearch(arr, 20, 4)
5	4	lsearch(arr, 20, 3)
6	3	lsearch(arr, 20, 2)
7	2	lsearch(arr, 20, 1)
8	1	lsearch(arr, 20, 0)
9	0	-1

Tail recursion

```
int lsearch(int arr[], int val, int len) {
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   return lsearch(arr, val, len-1);
}
```

The caller simply returns after the recursive call.

```
arr: 13, 79, 87, 32, 65, 23, 11, 57, 23
search : 20
```

```
int lsearch(int arr[], int val, int len) {
start:
   if (len == 0)
      return -1;
   if (arr[len-1] == val)
      return len-1;
   len = len – 1;
   goto start;
}
```

After eliminating recursive call.

Tail recursion

- An algorithm uses tail recursion if it simply returns after recursive calls without doing additional computation

- The compiler can automatically transform such algorithms into iterative algorithms

- Therefore, the tail-recursive algorithms are equally efficient as their iterative counterparts

Recursion vs iteration

- Recursive programs usually turn out to be concise and elegant
- Recursive solutions can quickly get extremely slow if not designed carefully
- In most cases, the iterative solution works faster than the recursive counterpart
- Tail recursion can eliminate the overhead of additional calls
- However, there are many problems for which tail-recursive solutions do not exist

Efficiency of algorithms

References

- Chapter-4 from Goodrich, Tamassia and Mount book

Fibonacci numbers (algorithm-1)

- Recursive solution

```
int fib(int n) {
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

Fibonacci numbers (algorithm-2)

- Iterative solution

n = 5.
f_1 = 1
f_2 = 0
i = 2
f_n = 1
f_n_2 = 1
f_n_1 = 1

i = 3
f_n = 2
f_n_2 = 1
f_n_1 = 2

i = 4
f_n = 3
f_n_2 = 2
f_n_1 = 3

i = 5
f_n = 5
f_n_2 = 3
f_n_1 = 5

0 1 1 2 3 5

5

```
int fib(int n) {
  int f_n_minus_1 = 1;
  int f_n_minus_2 = 0;
  int f_n, i;

  if (n < 2) {
    return n;
  }

  for (i = 2; i <= n; i++) {
    f_n = f_n_minus_1 + f_n_minus_2;
    f_n_minus_2 = f_n_minus_1;
    f_n_minus_1 = f_n;
  }
  return f_n;
}
```

Fibonacci numbers

- A better solution

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix} = \begin{bmatrix} f(n-1) + f(n-2) \\ f(n-1) + 0 \end{bmatrix}$$

Fibonacci numbers

- A better solution

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} f(n-2) \\ f(n-3) \end{bmatrix}$$

Fibonacci numbers

- A better solution

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} f(n-2) \\ f(n-3) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^3 \begin{bmatrix} f(n-3) \\ f(n-4) \end{bmatrix} = \dots = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} f(1) \\ f(0) \end{bmatrix}$$

$$= \underline{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}}$$

Fibonacci numbers

- Let $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

$$f(n) = A^{n-1} * \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

How to compute A^{n-1} fast?

Fibonacci numbers (algorithm-3)

$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

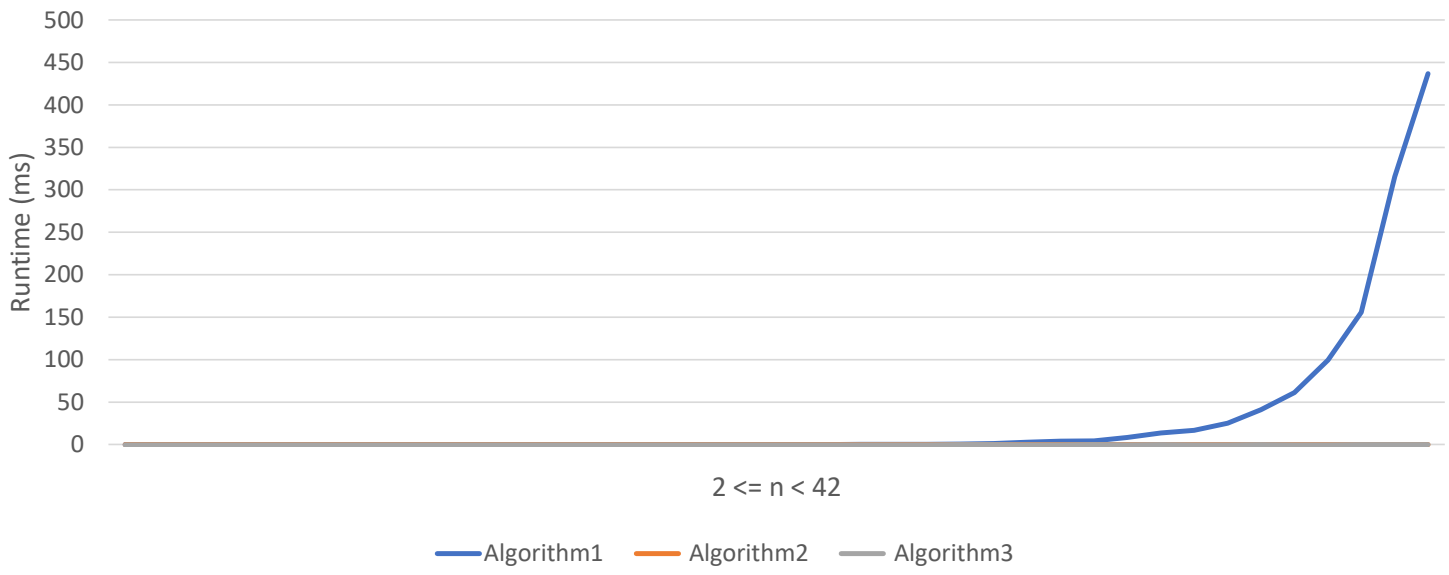
```
// mul takes a matrix A and R as input
// returns the result of Aⁿ in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

```
int fib(int n) {
  if (n <= 1) {
    return n;
  }
  int A[2][2] = {{1, 1}, {1, 0}};
  int R[2][2];
  mul(A, R, n-1);
  // R contains A^(n-1)
  return R[0][0];
}
```

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} R[0][0] & R[0][1] \\ R[1][0] & R[1][1] \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} R[0][0] \\ R[1][0] \end{bmatrix}$$

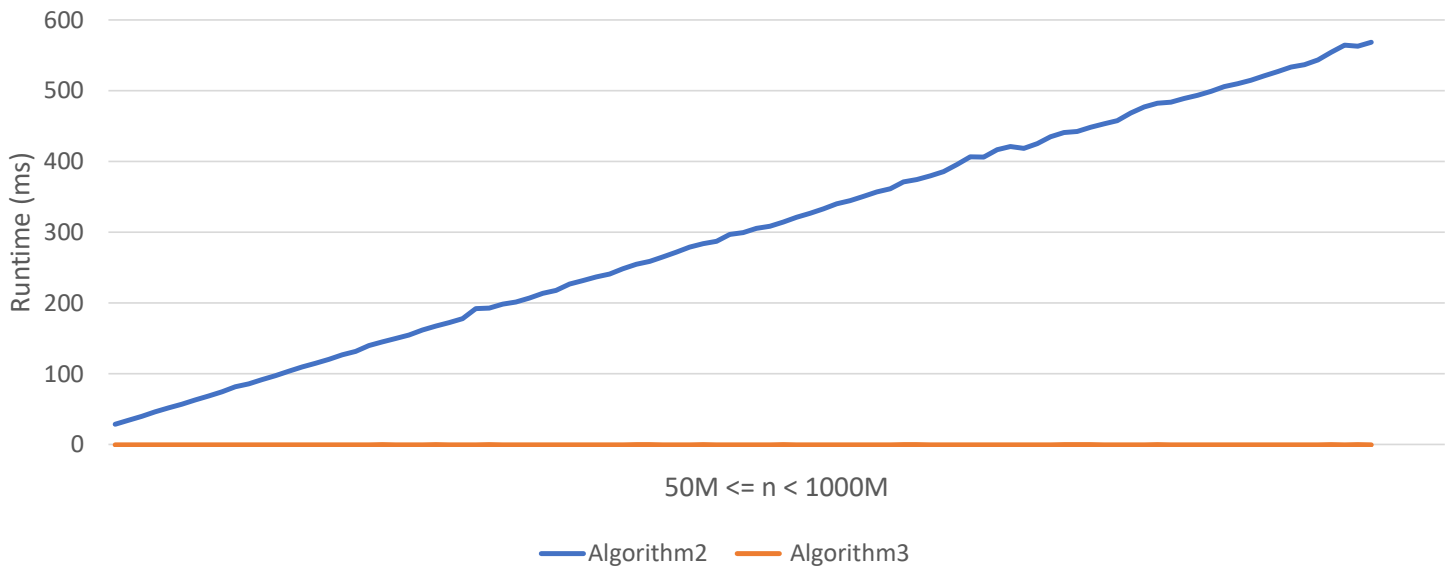
Runtime statistics

- For various number of n



Runtime statistics

- For various number of n



Analysis of algorithms

- Experiments are performed on a set of test inputs that may not include the all possible behaviors
- The algorithm may perform differently on different platforms
- The algorithm needs to be fully implemented before the runtime analysis

An example

```
int example(int arr[], int n) {
  int ret = -1;
  if (n > 1000) {
     if (n > 5000) {
       ret = sort(arr, n);
     } else if (n > 3000) {
       ret = set_zero(arr, n);
     } else {
       ret = search_zero(arr, n);
     }
  } else {
     if (n > 900) {
       ret = sum(arr, n);
     } else if (n > 700) {
       ret = reverse(arr, n);
     } else {
       ret = sum_all(arr, n, 2);
     }
  }
  return ret;
}
```

Generate a set of test inputs needed to cover all the paths in the example routine at runtime.

6000, 4000, 2000,

990, 770,

600

Analysis of algorithms

- Even though we are successful in generating a set of test inputs that cover all program paths at runtime, the runtime when the program takes a given path may vary depending on the input
  - For example, the runtime of the sorting algorithm may depend on the size of the array and the contents of the array
    - e.g., some sorting algorithms may work faster if the input array is mostly sorted

add r1, r2
mov r3 (r)

Analysis of algorithms

- Time taken by an algorithm is directly proportional to the sum of the costs of CPU instructions executed at runtime
  - Counting the number of CPU instructions may be a tedious job

- Alternatively, counting the number of high-level (primitive) operations is a good approximation

Primitive operations

- Assigning a value to a variable
- Calling a method
- Performing an arithmetic operation
- Comparing two numbers
- Indexing into an array
- Returning from a method

Fibonacci numbers (algorithm-1)

- Recursive solution

n	Num operations
≤1	2
>1	7

```
int fib(int n) {
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

Fibonacci numbers (algorithm-2)

- Iterative solution

n	Num operations
<2	4
>=2	3 + (n-1)*6 + 1

```
int fib(int n) {
  int f_n_minus_1 = 1;
  int f_n_minus_2 = 0;
  int f_n, i;

  if (n < 2) {
    return n;
  }

  for (i = 2; i <= n; i++) {
    f_n = f_n_minus_1 + f_n_minus_2;
    f_n_minus_2 = f_n_minus_1;
    f_n_minus_1 = f_n;
  }
  return f_n;
}
```

Fibonacci numbers (algorithm-3)

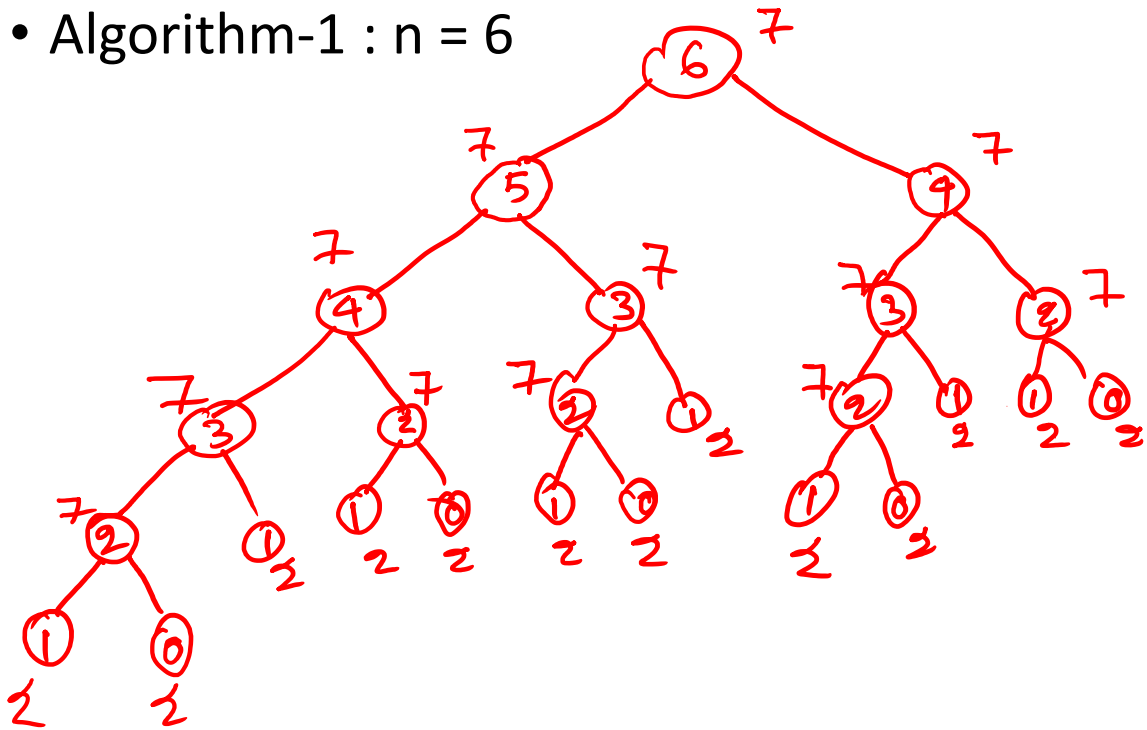
```
// mul takes a matrix A and R as input
// returns the result of Aⁿ in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

```
int fib(int n) {
  if (n <= 1) {
    return n;
  }
  int A[2][2] = {{1, 1}, {1, 0}};
  int R[2][2];
  mul(A, R, n-1);
  // R contains A^(n-1)
  return R[0][0];
}
```

n	Num operations
=1	14
mul2	41
n%2 = 0	46
n%2 = 1	88
fib	13

Fibonacci numbers

- Algorithm-1 : n = 6



n	Num operations
≤1	2
>1	7

$13 \times 2 + 7 \times 12$

$= 26 + 84$

$= 110$

Fibonacci numbers

- Algorithm-2: n = 128

$3 + 127 \times 6 + 1$

$4 + 762 = 766$

n	Num operations
<2	4
>=2	3 + (n-1)*6 + 1

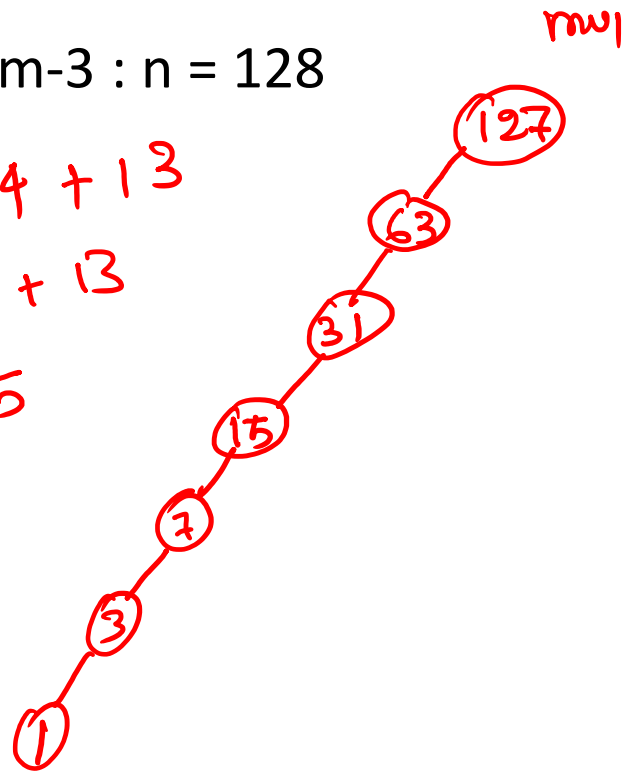
Fibonacci numbers

- Algorithm-3 : n = 128

6 * 88 + 14 + 13

528 + 14 + 13

= 555



n	Num operations
=1	14
mul2	41
n%2 = 0	46
n%2 = 1	88
fib	13

Number of operations

- Algorithm-1 ($f(n) = f(n-1) + f(n-2)$)
  - $2^{\frac{n-1}{2}} < num\ operations < 2^{n-1}$

- Algorithm-2 (iterative)
  - $num\ operations = 6\mathrm{n} - 2$

- Algorithm-3 (matrix multiplication)
  - $c_1 log_2 n \leq \mathrm{num\ operations} \leq \mathrm{c_2 log_2 n}$
    - for some constants c_1 and c_2

Time complexity

- The number of primitive operations in the worst case as a function of input size
  - e.g., the time complexity of the iterative algorithm for Fibonacci is $6n - 2$

Today’s lecture

- Time complexity
- Big-Oh notation
- Recurrence relation

Fibonacci numbers (algorithm-1)

- Recursive solution

```
int fib(int n) {
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

Fibonacci numbers (algorithm-2)

- Iterative solution

```
int fib(int n) {
  int f_n_minus_1 = 1;
  int f_n_minus_2 = 0;
  int f_n, i;

  if (n < 2) {
    return n;
  }

  for (i = 2; i <= n; i++) {
    f_n = f_n_minus_1 + f_n_minus_2;
    f_n_minus_2 = f_n_minus_1;
    f_n_minus_1 = f_n;
  }
  return f_n;
}
```

Fibonacci numbers (algorithm-3)

```
// mul takes a matrix A and R as input
// returns the result of Aⁿ in R
void mul(int A[2][2], int R[2][2], int n) {
  if (n == 1) {
    R[0][0] = A[0][0];  R[0][1] = A[0][1];
    R[1][0] = A[1][0];  R[1][1] = R[1][1];
    return;
  }
  if (n % 2 == 0) {
    mul(A, R, n/2);
    // mul2 takes two 2x2 matrices as input and
    // returns the multiplication in the first
    // matrix
    mul2(R, R);  // R <- R * R
  }
  else {
    mul(A, R, (n-1)/2);
    mul2(R, R);  // R <- R * R
    mul2(R, A);  // R <- R * A
  }
}
```

```
int fib(int n) {
  if (n <= 1) {
    return n;
  }
  int A[2][2] = {{1, 1}, {1, 0}};
  int R[2][2];
  mul(A, R, n-1);
  // R contains Aⁿ⁻¹
  return R[0][0];
}
```

Primitive operations

- Assigning a value to a variable
- Calling a method
- Performing an arithmetic operation
- Comparing two numbers
- Indexing into an array
- Returning from a method

Number of operations

- Algorithm-1 ($f(n) = f(n-1) + f(n-2)$)
  - $2^{\frac{n-1}{2}} < num\ operations < 2^{n-1}$
- Algorithm-2 (iterative)
  - $num\ operations = 6\mathrm{n} - 2$
- Algorithm-3 (matrix multiplication)
  - $c_1 log_2 n \leq \mathrm{num\ operations} \leq \mathrm{c_2 log_2 n}$
    - for some constants c_1 and c_2

Time complexity

- The number of primitive operations in the worst case as a function of input size
  - e.g., the time complexity of the iterative algorithm for Fibonacci is $6n - 2$

Time complexity

- Does counting the number of primitive operations give us the actual time complexity?

Time complexity

- Does counting the number of primitive operations give us the actual time complexity?
  - Does calling a function is same as a variable assignment?

- Can we simplify the analysis further?

Time complexity

- Does counting the number of primitive operations give us the actual time complexity?
  - Does calling a function is same as a variable assignment?

- Can we simplify the analysis further?
  - Instead of finding the real complexity, we try to find the lower and upper bounds of the time complexity
  - We try to make lower and upper bounds close to the actual complexity

Time complexity

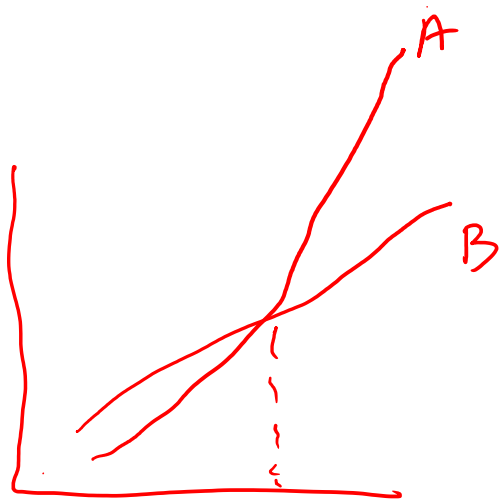
- We will discuss a general method to compute time complexity, which
  - Considers all possible inputs
  - Can be used to compare two algorithms in a way that is independent of underline hardware and software platform
  - Can be performed on the pseudo-code of an algorithm

Which algorithm is better?

$A = 3n^2 + 2$
$B = 8n^2 + 3n + 10$

Which algorithm is better?

$A = n^2$
$B = 10n + 1$



Which algorithm is better?

$A = n^2$
$B = 10n + 1$

- Time complexity matters for large input size

Which algorithm is better?

$A = n^2 + 1$
$B = n^2 + 1000n + 10000$

Which algorithm is better?

$$A = n^2 + 1$$
$$B = n^2 + 1000n + 10000$$

n	A = n^2 + 1	B = n^2 + 1000n + 10000	A/B
1000	1000001	2010000	0.497513
10000	100000001	1.1E+08	0.909008
100000	1E+10	1.01E+10	0.990098
1000000	1E+12	1E+12	0.999001
10000000	1E+14	1E+14	0.9999

Which algorithm is better?

$$A = n^2 + 1$$
$$B = n^2 + 1000n + 10000$$

n	A = n^2 + 1	B = n^2 + 1000n + 10000	A/B
1000	1000001	2010000	0.497513
10000	100000001	1.1E+08	0.909008
100000	1E+10	1.01E+10	0.990098
1000000	1E+12	1E+12	0.999001
10000000	1E+14	1E+14	0.9999

For large n, only the highest order term in the time complexity matters.

Which algorithm is better?

$$A = 3n^2 + 1$$
$$B = 7n^2 + 1000n + 10000$$
$$C = 100n^{1.5} + 2000n + 20000$$

n	A = $3n^2 + 1$	B = $7n^2 + 1000n + 10000$	C = $100n^{1.5} + 2000n + 20000$	A/B	C/A
1000	3000001	8010000	5182277.7	0.374532	1.727425
10000	3E+08	7.1E+08	120020000	0.422529	0.400067
100000	3E+10	7.01E+10	3.362E+09	0.42796	0.112077
1000000	3E+12	7E+12	1.02E+11	0.42851	0.034
10000000	3E+14	7E+14	3.182E+12	0.428565	0.010608
100000000	3E+16	7E+16	1.002E+14	0.428571	0.00334

Which algorithm is better?

$$A = 3n^2 + 1$$
$$B = 7n^2 + 1000n + 10000$$
$$C = 100n^{1.5} + 2000n + 20000$$

n	A = $3n^2 + 1$	B = $7n^2 + 1000n + 10000$	C = $100n^{1.5} + 2000n + 20000$	A/B	C/A
1000	3000001	8010000	5182277.7	0.374532	1.727425
10000	3E+08	7.1E+08	120020000	0.422529	0.400067
100000	3E+10	7.01E+10	3.362E+09	0.42796	0.112077
1000000	3E+12	7E+12	1.02E+11	0.42851	0.034
10000000	3E+14	7E+14	3.182E+12	0.428565	0.010608
100000000	3E+16	7E+16	1.002E+14	0.428571	0.00334

Algorithm A is better than B. However, the improvement factor remains constant for large input sizes.

Which algorithm is better?

$$A = 3n^2 + 1$$
$$B = 7n^2 + 1000n + 10000$$
$$C = 100n^{1.5} + 2000n + 20000$$

n	A = $3n^2 + 1$	B = $7n^2 + 1000n + 10000$	C = $100n^{1.5} + 2000n + 20000$	A/B	C/A
1000	3000001	8010000	5182277.7	0.374532	1.727425
10000	3E+08	7.1E+08	120020000	0.422529	0.400067
100000	3E+10	7.01E+10	3.362E+09	0.42796	0.112077
1000000	3E+12	7E+12	1.02E+11	0.42851	0.034
10000000	3E+14	7E+14	3.182E+12	0.428565	0.010608
100000000	3E+16	7E+16	1.002E+14	0.428571	0.00334

Algorithm C is a huge improvement over A because the improvement factor (lower is better) approaches zero as the input size grows. In other words, the improvement is directly proportional to the input size.

Efficiency of algorithms

- Observations so far
  - We analyze the algorithms for **large input size**
  - Only the **highest degree term** in the time complexity matters
  - We disregard the constant factors in the formula and focus on the **huge-improvement** aspects of the algorithm

Asymptotic notation

(Captures the observations we have made so far)

Big-Oh notation

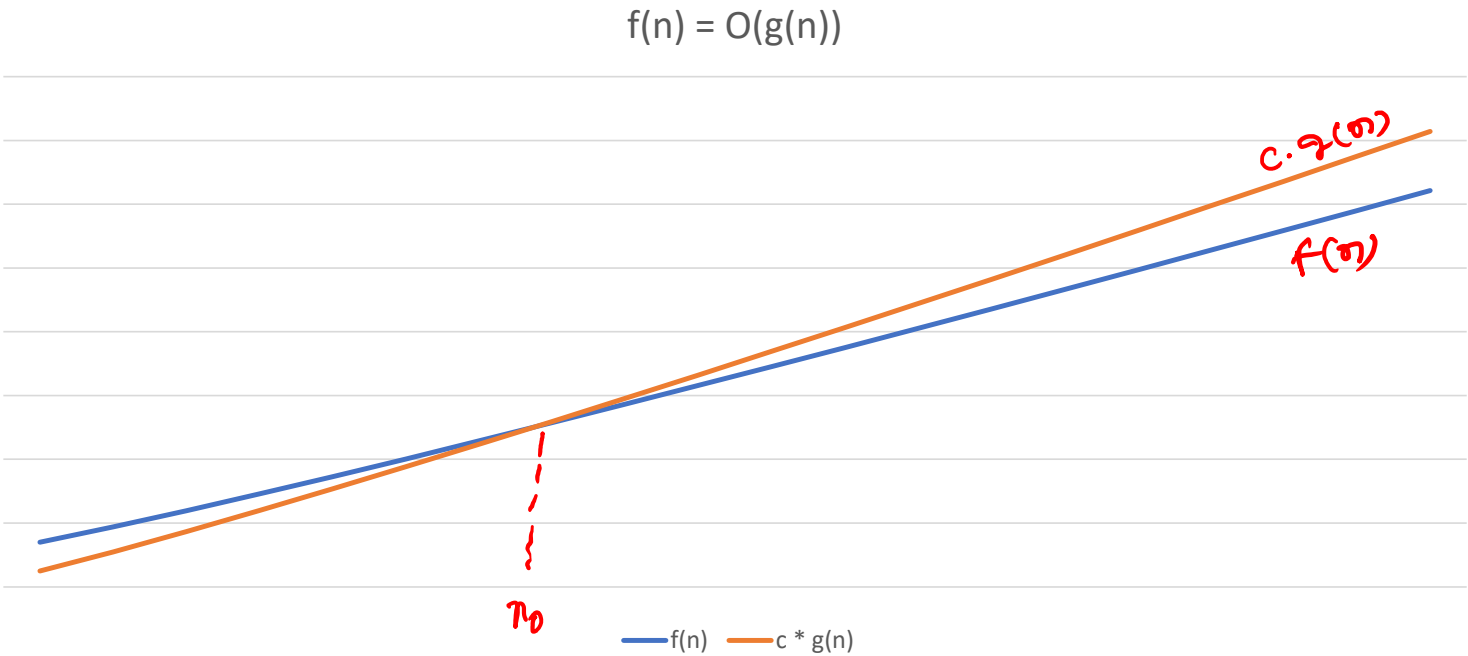
Let $f(n)$ and $g(n)$ are increasing functions of n

$f(n)\ is\ O\big(g(n)\big)$ (pronounced as order of)

if for some constants $c > 0$ and $n_0 \geq 1$

$$f(n) \leq cg(n), \qquad for\ all\ n > n_0$$

Big-Oh notation



c * g(n) is an asymptotic upper bound for f(n)

Example

$8n^4 + 2n^3 + n^2 + 3$ *is* $O(n^4)$

$8n^4 + 2n^3 + n^2 + 3 \leq c\,n^4 \qquad n_0 > 0$

$8n^4 + 2n^3 + n^2 + 3 \leq 8n^4 + 2n^4 + n^4 + 3n^4$

$O(n^5)$

$O(8n^4 + 2n^3 + n^2 + 3)$

Example

$a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_k n^k$ is $O(n^k)$ if $a_k > 0$

Example

$5n^2 + 8nlog_2n + 3$ $= O(n^2)$ $\leq 16n^2$

$3nlog_2n + n$ $= O(n\log n)$

$10n + 3\log_2 n$ $= O(n)$

100 $= O(1)$

$100 * 2^{n+12}$ $= O(2^n)$

2^{1000} $= O(1)$

Big-Oh notation

- Big-Oh is used to describe asymptotic upper bound

- Even though there are many possibilities for the upper bound, we try to stay close to the real complexity
  - e.g., $3n^2 + 8 \; is \; O(n^4)$ but doesn't make much sense

- We remove the constant factors and lower order terms in the big-Oh notation

Space complexity

- Similar to the time complexity, we can also measure space complexity
- The space complexity tells us about the memory utilization of the program in the worst case as a function of input size
- For most algorithms, space complexity is not a big concern

Analysis of algorithm

- Time complexity
  - Predict how much time an algorithm will take for a given input size

- Space complexity
  - Predict how much space an algorithm will take for a given input size

Linear search

```
int lsearch(int arr[], int val, int n) {
   int i;
   for (i = n-1; i >= 0; i--) {
      if (arr[i] == val)
         return i;
   }
   return -1;
}
```

c

Time complexity:

$nc + c_1$

$O(n)$

Selection sort

```
void selection_sort(int arr[], int n)
{
  int i, idx;

  for (i = 0; i < n - 1; i++) {
    idx = find_min(arr, i, n);
    if (idx != i) {
      swap(arr, i, idx);
    }
  }
}
```

Time complexity:

$(nc_1 + c_2) + (n-1)c_1 + c_2$
$+ (n-2)c_1 + c_2 + \ldots + 2c_1 + c_2$

```
int find_min(int arr[], int start, int end)
{
  int i, min, idx;

  min = arr[start];
  idx = start;
  for (i = start + 1; i < end; i++) {
    if (arr[i] < min) {
      min = arr[i];
      idx = i;
    }
  }
  return idx;
}

void swap(int arr[], int i, int j)
{
  int tmp = arr[i];
  arr[i] = arr[j];
  arr[j] = tmp;
}
```

Time complexity

$$(nc_1 + c_2) + ((n-1)c_1 + c_2) + ((n-2)c_1 + c_2) + \dots + (2c_1 + c_2)$$

$$= c_1(2+3+\dots+n) + c_2(n-1)$$

$$= c_1\left(\frac{n(n+1)}{2} - 1\right) + c_2(n-1)$$

$$= \frac{c_1 n^2}{2} + \frac{c_1 n}{2} - c_1 + c_2 n - c_2$$

$$1+2+3+\dots+n = \frac{n(n+1)}{2}$$

$$O(n^2)$$

Time complexity

$$
\begin{aligned}
T(n) = & \ (c_1 n + c_2) + (c_1(n-1) + c_2) + (c_1(n-3) + c_2) + \ \ldots + (c_1 * 2 + c_2) \\
& = c_1(2 + 3 + 4 + \ \ldots + n) + c_2(n-1) \\
& = c_1\left(\frac{n(n+1)}{2} - 1\right) + c_2(n-1) \\
& = \frac{c_1 n^2}{2} + \frac{c_1 n}{2} - c_1 + c_2 n \ - c_2 \\
& = O(n^2)
\end{aligned}
$$

Fibonacci numbers

```
int fib(int n) {
  int f_n_minus_1 = 1;
  int f_n_minus_2 = 0;
  int f_n, i;

  if (n < 2) {
    return n;
  }

  for (i = 2; i <= n; i++) {
    f_n = f_n_minus_1 + f_n_minus_2;
    f_n_minus_2 = f_n_minus_1;
    f_n_minus_1 = f_n;
  }
  return f_n;
}
```

Time complexity:

$(n-1)\,c_1 + c_2$

$O(n)$

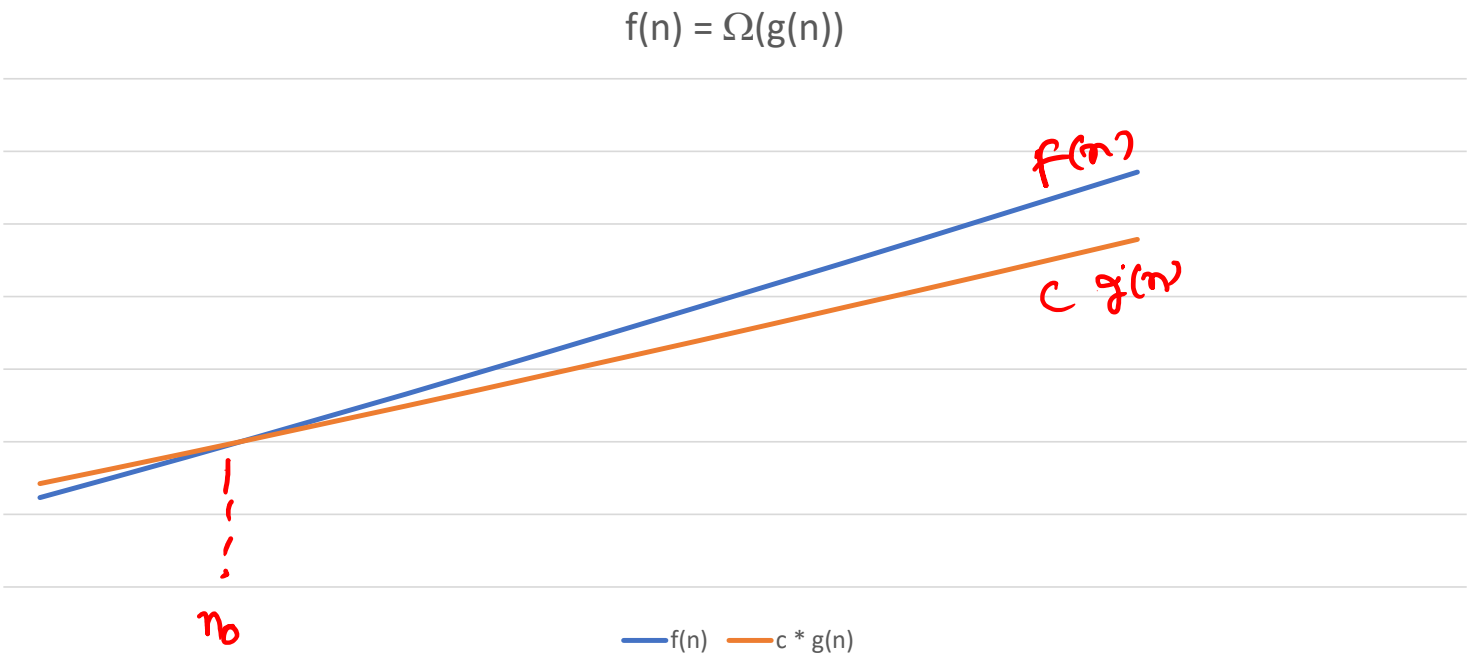
Big-Omega notation

Let $f(n)$ and $g(n)$ are increasing functions of n

$f(n)\ is\ \Omega\big(g(n)\big)$ (pronounced as big-Omega of)
if for some constants $c > 0$ and $n_0 \geq 1$

$$f(n) \geq cg(n), \qquad for\ all\ n > n_0$$

Big-Omega notation



c * g(n) is an asymptotic lower bound for f(n)

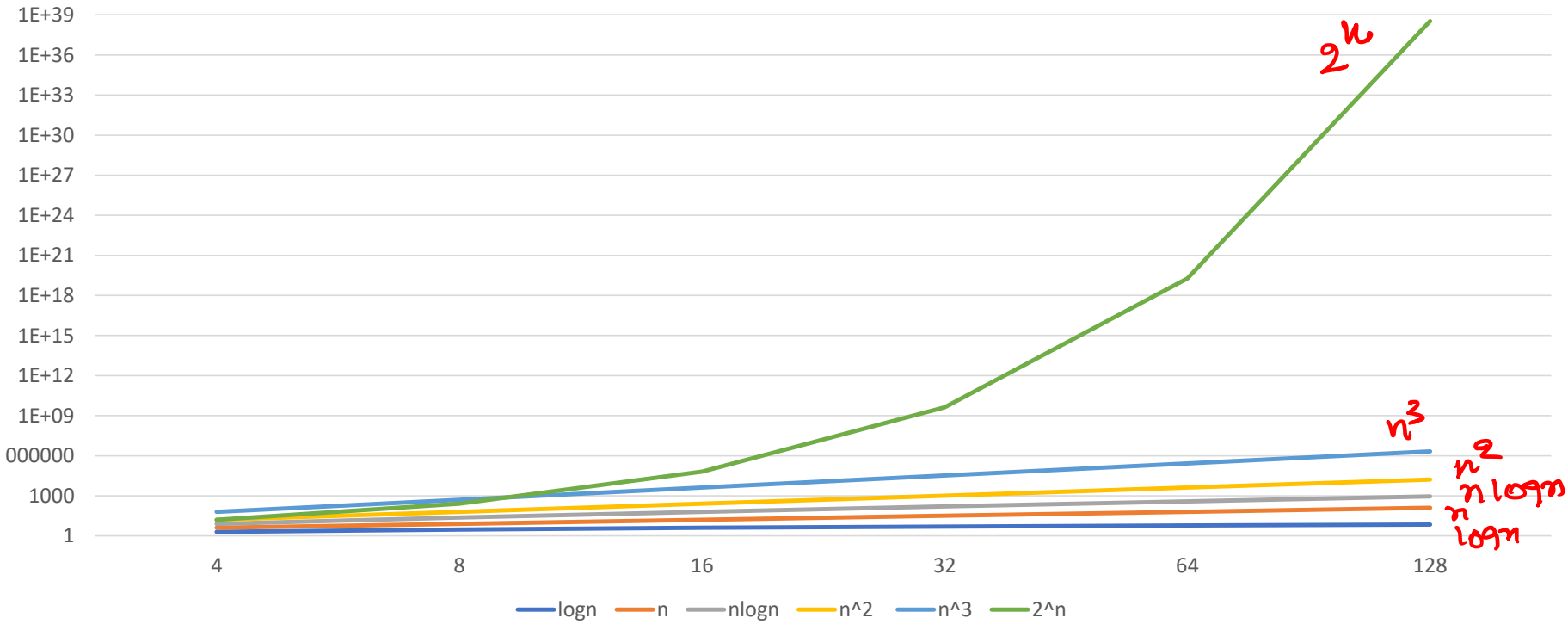
Big-Omega

$3nlogn + 2n$ $= \Omega(n\log n)$

$3n^3 + 2n^2 + 10$ $= \Omega(n^3)$ $\Omega(n^2)$

1000 $= \Omega(1)$

Growth of functions



Problem size (n) vs Time complexity

Complexity	1 second	1 minute	1 hour
n	60M	360M	21,600M
nlogn	2M	113M	5,740M
n^2	7,746	60,000	4,64,758
n^3	392	1534	6000
2^n	26	32	38

26+8

256

Common complexities

	Complexity	Common name
Best	O(1)	Constant
	O(logn)	Logarithmic
	O(n)	Linear
	O(nlogn)	n-log-n
	$O(n^2)$	Quadratic
	$O(n^3)$	Cubic
Worst	$O(2^n)$	Exponential

Recurrence relation

Recurrence relation

- A recurrence relation is an equation or inequality that expresses the nth term of a sequence as a function of the k preceding terms
  - k is called the order of the recurrence relation
  - e.g., `f(n) = f(n-1) + f(n-2)` is a recurrence relation

- Source: Wikipedia

References

- Read section-3.5 from the Udi Manber book

Solving recurrence relation

- Expansion method
- Intelligent guesses
- Elimination of history
  - Full-history recurrence relation
    - Depends on all preceding terms instead of a few

Expansion method

- STEP-1 : Expand and simplify the recurrence

```
T(1) = 1
T(n) = T(n-1) + c
     = (T(n-2) + c) + c      // expand
     = T(n-2) + 2c           // simplify
     = (T(n-3) + c) + 2c     // expand
     = T(n-3) + 3c           // simplify
```

Expansion method

- STEP-2: Identify a pattern after the kth iteration

```
T(1) = 1
T(n) = T(n-1) + c
     = (T(n-2) + c) + c     // expand
     = T(n-2) + 2c          // simplify
     = (T(n-3) + c) + 2c    // expand
     = T(n-3) + 3c          // simplify
     …
     = T(n-k) + kc          // after the kth iteration
```

Expansion method

- STEP-3: Substitute k with a value such that T(n) can be expressed as a function of base cases

```
T(1) = 1
T(n) = T(n-1) + c
     = (T(n-2) + c) + c     // expand
     = T(n-2) + 2c          // simplify
     = (T(n-3) + c) + 2c    // expand
     = T(n-3) + 3c          // simplify
     …
     = T(n-k) + kc          // after the kth iteration

Substitute k with n-1
T(n) = T(1) + c(n-1)
```

Expansion method

- STEP-4: Simplify the equation

```
T(1) = 1
T(n) = T(n-1) + c
     = (T(n-2) + c) + c     // expand
     = T(n-2) + 2c          // simplify
     = (T(n-3) + c) + 2c    // expand
     = T(n-3) + 3c          // simplify
     …
     = T(n-k) + kc          // after the kth iteration

Substitute k with n-1
T(n) = T(1) + c(n-1)
     = 1 + cn – c          // no further simplification
```

Expansion method

```
T(1) = 1
T(n) = T(n-1) + c
```

$$T(n) = T(n-1) + c$$
$$= (T(n-2) + c) + c$$
$$= T(n-2) + 2c$$
$$= (T(n-3) + c) + 2c$$
$$= T(n-3) + 3c$$
$$\cdots$$
$$= T(n-k) + kc$$

$$k = n-1$$
$$T(n) = T(1) + c(n-1)$$
$$= 1 + cn - c \quad = O(n)$$

Time complexity

- For recursive programs, time complexities can be expressed as recurrence relations

Factorial

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

Time complexity:

$T(n) = T(n-1) + c$

$T(0) = 2$

Time complexity

$T(0) = 2$

$$
\begin{aligned}
T(n) &= T(n-1) + c \\
&= (T(n-2) + c) + c = T(n-2) + 2c \\
&= (T(n-3) + c) + 2c = T(n-3) + 3c \\
&= \cdots \\
&= T(n - k) + c\mathrm{k}
\end{aligned}
$$

Substitute, k = n

$$
\begin{aligned}
&= T(0) + cn = 2 + \mathrm{cn} \\
&= O(n)
\end{aligned}
$$

Linear search

```
int lsearch(int arr[], int val, int n) {
   if (n == 0)
      return -1;
   if (arr[n-1] == val)
      return n-1;
   return lsearch(arr, val, n-1);
}
```

Time complexity:

$T(n) = T(n-1) + c$

$T(0) = 2$

$O(n)$

Binary search

```
int bsearch(int arr[], int val, int lo, int hi) {
   if (hi < lo)
      return -1;
   int mid = (lo + hi) / 2;
   if (arr[mid] == val)
      return mid;
   if (arr[mid] > val)
      return bsearch(arr, val, lo, mid-1);
   else
      return bsearch(arr, val, mid+1, hi);
}
```

$\frac{\log_2 n}{\log_{10} 2} + \frac{\log_2 8^2}{\log_2}$

$O(\log n)$

Time complexity: $T(1) = c_1$

$$T(n) = T(n/2) + c$$
$$= \left(T\left(\frac{n}{4}\right) + c\right) + c$$
$$= T\left(\frac{n}{2^2}\right) + 2c$$
$$= \left(T\left(\frac{n}{2^3}\right) + c\right) + 2c$$
$$= T\left(\frac{n}{2^3}\right) + 3c$$
$$\vdots$$
$$= T\left(\frac{n}{2^k}\right) + kc$$
$$= T\left(\frac{n}{n}\right) + \log_2 n \cdot c$$
$$= c_1 + c\log_2 n$$

substitute $n = 2^k$

$\log_2 n = k$

Time complexity

$T(1) = c_1$

$$T(n) = T\left(\frac{n}{2}\right) + c$$
$$= \left(T\left(\frac{n}{2^2}\right) + c\right) + c = T\left(\frac{n}{2^2}\right) + 2c$$

$$= \left(T\left(\frac{n}{2^3}\right) + c\right) + 2c = T\left(\frac{n}{2^3}\right) + 3c$$
$$= \ \ldots$$

$$= T\left(\frac{n}{2^k}\right) + kc$$

Substitute $n = 2^k$
$T(n) = T(1) + clog_2(n) = O(\log(n))$

Selection sort

```
void selection_sort_r(int arr[], int pos, int n) {
  int idx;

  if (pos >= n - 1) {
    return;
  }

  idx = find_min(arr, pos, n);
  if (idx != pos) {
    swap(arr, pos, idx);
  }
  selection_sort(arr, pos+1, n);
}
```

Time complexity:

$T(1) = c$

$T(n) = T(n-1) + nc_1 + c_2$

```
int find_min(int arr[], int start, int end)
{
  int i, min, idx;

  min = arr[start];
  idx = start;
  for (i = start + 1; i < end; i++) {
    if (arr[i] < min) {
      min = arr[i];
      idx = i;
    }
  }
  return idx;
}

void swap(int arr[], int i, int j)
{
  int tmp = arr[i];
  arr[i] = arr[j];
  arr[j] = tmp;
}
```

Time complexity

$T(1) = c$

$T(n) = T(n-1) + nc_1 + c_2$

$= (T(n-2) + (n-1)c_1 + c_2) + nc_1 + c_2$

$= T(n-2) + (n-1)c_1 + nc_1 + 2c_2$

$= (T(n-3) + (n-2)c_1 + c_2) + (n-1)c_1 + nc_1 + 2c_2$

$= T(n-3) + (n-2)c_1 + (n-1)c_1 + nc_1 + 3c_2$

$\cdots\cdots$

$= T(n-k) + (n-k+1)c_1 + \cdots + (n-2)c_1 + (n-1)c_1 + nc_1 + kc_2$

$k = n-1$

$T(n) = T(1) + 2c_1 + \cdots + (n-2)c_1 + (n-1)c_1 + nc_1 + (n-1)c_2$

$= c + c_1(2+3+\cdots+n) + c_2(n-1)$

$= c + c_1\left(\frac{n(n+1)}{2} - 1\right) + c_2(n-1)$

$= O(n^2)$

Time complexity

$T(1) = c$

$T(n) = T(n-1) + c_1 n + c_2$

$= (T(n-2) + c_1(n-1) + c_2) + c_1 n + c_2$

$= T(n-2) + c_1(n-1) + c_1 n + 2c_2$

$= (T(n-3) + c_1(n-2) + c_2) + c_1(n-1) + c_1 n + 2c_2$

$= T(n-3) + c_1(n-2) + c_1(n-1) + c_1 n + 3c_2$

$= \ \ldots$

$= T(n-k) + c_1(n-k+1) + \cdots + c_1(n-2) + c_1(n-1) + c_1 n + kc_2$

Substitute, k = n-1

$= T(1) + 2c_1 + \ \ldots + c_1(n-2) + c_1(n-1) + c_1 n + c_2(n-1)$

$= c + c_1(2 + 3 + \cdots + n) + c_2(n-1)$

$= c + c_1\left(\frac{n(n+1)}{2} - 1\right) + c_2(n-1)$

$= O(n^2)$

Tower of Hanoi

```
void move(int n, char src_t[], char dst_t[], char mid_t[]) {
  if (n == 0)
    return;
  move(n-1, src_t, mid_t, dst_t);
  printf("moving from %s to %s\n", src_t, dst_t);
  move(n-1, mid_t, dst_t, src_t);
}

int main() {
  move(4, "Tower1", "Tower3", "Tower2");
  return 0;
}
```

Time complexity: $T(0) = 2$

$T(n) = 2T(n-1) + c$

Time complexity

$T(0) = 2$

$$T(n) = 2T(n-1) + c$$
$$= 2(2T(n-2) + c) + c$$
$$= 2^2 T(n-2) + 2c + c$$
$$= 2^2(2T(n-3) + c) + 2c + c$$
$$= 2^3 T(n-3) + 2^2 c + 2c + c$$

$\cdots$

$$= 2^k T(n-k) + 2^{k-1} c + \cdots + 2^2 c + 2c + c$$

$k = n$

$$= 2^n T(0) + c(1 + 2 + 2^2 + \cdots + 2^{n-1})$$
$$= 2^{n+1} + c(2^n - 1)$$

$O(2^n)$

Time complexity

$$T(0) = 2$$

$$\begin{aligned}
T(n) &= 2T(n-1) + c \\
&= 2(2T(n-2) + c) + c \\
&= 2^2T(n-2) + 2c + c \\
&= 2^2(2T(n-3) + c) + 2c + c \\
&= 2^3T(n-3) + 2^2c + 2c + c \\
&= \ \dots \\
&= 2^kT(n-k) + 2^{k-1}c + \ \dots + 2^2c + 2c + c
\end{aligned}$$

Substitute k = n

$$\begin{aligned}
&= 2^nT(n-n) + 2^{n-1}c + \ \dots + 2^2c + 2c + c \\
&= 2^nT(0) + c(1 + 2 + \cdots + 2^{n-1}) \\
&= 2^{n+1} + c(2^n - 1) \\
&= O(2^n)
\end{aligned}$$

Today’s topics

- Recurrence relation
- Merge sort
- Quick sort

Solving recurrence relation

- Expansion method
- Intelligent guesses
- Elimination of history
  - Full-history recurrence relation
    - Depends on all preceding terms instead of a few

Expansion method

- STEP-1 : Expand and simplify the recurrence

```
T(1) = 1
T(n) = T(n-1) + c
     = (T(n-2) + c) + c      // expand
     = T(n-2) + 2c           // simplify
     = (T(n-3) + c) + 2c     // expand
     = T(n-3) + 3c           // simplify
```

Expansion method

- STEP-2: Identify a pattern after the kth iteration

```
T(1) = 1
T(n) = T(n-1) + c
     = (T(n-2) + c) + c     // expand
     = T(n-2) + 2c          // simplify
     = (T(n-3) + c) + 2c    // expand
     = T(n-3) + 3c          // simplify
     …
     = T(n-k) + kc          // after the kth iteration
```

Expansion method

- STEP-3: Substitute k with a value such that T(n) can be expressed as a function of base cases

```
T(1) = 1
T(n) = T(n-1) + c
     = (T(n-2) + c) + c     // expand
     = T(n-2) + 2c          // simplify
     = (T(n-3) + c) + 2c    // expand
     = T(n-3) + 3c          // simplify
     …
     = T(n-k) + kc          // after the kth iteration

Substitute k with n-1
T(n) = T(1) + c(n-1)
```

Expansion method

- STEP-4: Simplify the equation

```
T(1) = 1
T(n) = T(n-1) + c
     = (T(n-2) + c) + c     // expand
     = T(n-2) + 2c          // simplify
     = (T(n-3) + c) + 2c    // expand
     = T(n-3) + 3c          // simplify
     …
     = T(n-k) + kc          // after the kth iteration

Substitute k with n-1
T(n) = T(1) + c(n-1)
     = 1 + cn – c          // no further simplification
```

Expansion method

```
T(1) = 1
T(n) = T(n-1) + c
```

$$T(n) = T(n-1) + c$$
$$= (T(n-2) + c) + c$$
$$= T(n-2) + 2c$$
$$= (T(n-3) + c) + 2c$$
$$= T(n-3) + 3c$$
$$\cdots$$
$$= T(n-k) + kc$$

$$k = n-1$$
$$T(n) = T(1) + c(n-1)$$
$$= 1 + cn - c \quad = O(n)$$

Binary search

```
int bsearch(int arr[], int val, int lo, int hi) {
   if (hi < lo)
      return -1;
   int mid = (lo + hi) / 2;
   if (arr[mid] == val)
      return mid;
   if (arr[mid] > val)
      return bsearch(arr, val, lo, mid-1);
   else
      return bsearch(arr, val, mid+1, hi);
}
```

$\frac{\log_2 n}{\log_{10} 2} + \frac{\log_2 8^2}{\log_2}$

$O(\log n)$

Time complexity: $T(1) = c_1$

$$T(n) = T(n/2) + c$$
$$= \left(T\left(\frac{n}{4}\right) + c\right) + c$$
$$= T\left(\frac{n}{2^2}\right) + 2c$$
$$= \left(T\left(\frac{n}{2^3}\right) + c\right) + 2c$$
$$= T\left(\frac{n}{2^3}\right) + 3c$$
$$\vdots$$
$$= T\left(\frac{n}{2^k}\right) + kc$$
$$= T\left(\frac{n}{n}\right) + \log_2 n \cdot c$$
$$= c_1 + c\log_2 n$$

substitute $n = 2^k$

$\log_2 n = k$

Time complexity

$T(1) = c$

$T(n) = T(n-1) + nc_1 + c_2$

$= (T(n-2) + (n-1)c_1 + c_2) + nc_1 + c_2$

$= T(n-2) + (n-1)c_1 + nc_1 + 2c_2$

$= (T(n-3) + (n-2)c_1 + c_2) + (n-1)c_1 + nc_1 + 2c_2$

$= T(n-3) + (n-2)c_1 + (n-1)c_1 + nc_1 + 3c_2$

$\cdots\cdots$

$= T(n-k) + (n-k+1)c_1 + \cdots + (n-2)c_1 + (n-1)c_1 + nc_1 + kc_2$

$k = n-1$

$T(n) = T(1) + 2c_1 + \cdots + (n-2)c_1 + (n-1)c_1 + nc_1 + (n-1)c_2$

$= c + c_1(2+3+\cdots+n) + c_2(n-1)$

$= c + c_1\left(\frac{n(n+1)}{2} - 1\right) + c_2(n-1)$

$= O(n^2)$

Time complexity

$T(0) = 2$

$$T(n) = 2T(n-1) + c$$
$$= 2(2T(n-2) + c) + c$$
$$= 2^2 T(n-2) + 2c + c$$
$$= 2^2(2T(n-3) + c) + 2c + c$$
$$= 2^3 T(n-3) + 2^2 c + 2c + c$$

$\cdots$

$$= 2^k T(n-k) + 2^{k-1} c + \cdots + 2^2 c + 2c + c$$

$k = n$

$$= 2^n T(0) + c(1 + 2 + 2^2 + \cdots + 2^{n-1})$$
$$= 2^{n+1} + c(2^n - 1)$$

$O(2^n)$

Master theorem

A recurrence relation of the form

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k, where\ a \geq 1, b \geq 2, c \geq 0, k \geq 0$$

has the following solution

$$T(n) = \begin{cases} O(n^{\log_b a}) & if\ a > b^k \\ O(n^k logn) & if\ a = b^k \\ O(n^k) & if\ a < b^k \end{cases}$$

Useful formulae

$$1 + 2 + 3 + \ \dots + n = \frac{n(n+1)}{2}$$

$$1 + r + r^2 + \ \dots + r^{n-1} = \frac{r^n - 1}{r - 1} \qquad if\ r\ \neq 1$$

$$1^2 + 2^2 + 3^2 + \ \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$1 + \frac{1}{2} + \frac{1}{3} + \ \dots + \frac{1}{n} \approx \ln x$$

Useful formulae

$$\log_b a = \frac{1}{\log_a b}$$

$$\log_a x = \frac{\log_b x}{\log_b a}$$

$$b^{\log_b x} = x$$

$$b^{\log_a x} = x^{\log_a b}$$

Useful formulae

$$f(x) = ax^2 + bx + c \quad where\ a, b, c \in R\ and\ a\ \neq 0$$

The roots of the above quadratic equation are:

$$\alpha = \frac{-b+\sqrt{b^2-4ac}}{2a} \qquad \beta = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

Fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

Time complexity:

$$T(n) = T(n-1) + T(n-2) + c$$

$$T(1) = c_1$$
$$T(0) = c_2$$

$$T(n-1) = T(n-2) + T(n-3) + c$$
$$T(n-2) = T(n-3) + T(n-4) + c$$

$$T(n) = T(n-2) + 2T(n-3) + T(n-4) + 2c$$

$$= \left(T(n-3) + T(n-4) + c\right) + 2\left(T(n-4) + T(n-5) + c\right) + \left(T(n-5) + T(n-6) + c\right) + 2c$$

$$= T(n-3) + 3T(n-4) + 3T(n-5) + T(n-6) + 6c$$

...

Time complexity

$T(1) = c_1$

$$T(n) = T(n-1) + T(n-2) + c$$

$$\leq 2T(n-1) + c$$

$$= 2(2T(n-2) + c) + c$$

$$= 2^2 T(n-2) + 2c + c$$

$$= 2^2(2T(n-3) + c) + 2c + c$$

$$= 2^3 T(n-3) + 2^2 c + 2c + c$$

$$\vdots$$

$$= 2^k T(n-k) + c(1 + 2 + 2^2 + \dots + 2^{k-1})$$

Substitute $k = n-1$

$$= 2^{n-1} T(1) + c(2^k - 1)$$

$$= c_1 2^{n-1} + c 2^{n-1} - c \qquad = O(2^n)$$

$$T(n) < c_2 2^{n-1} - c_3$$

Time complexity

$T(1) = c_1$

$$T(n) = T(n-1) + T(n-2) + c$$

$$T(n) > 2T(n-2) + c$$

$$T(n-2) > 2T(n-4) + c$$

$$T(n) > 2(2T(n-4) + c) + c$$

$$= 2^2 T(n-4) + 2c + c$$

$$= 2^2(2T(n-6) + c) + 2c + c$$

$$= 2^3 T(n - 2\times 3) + 2^2 c + 2c + c$$

$$\cdots$$

$$= 2^k T(n - 2\times k) + c(1 + 2 + 2^2 + \cdots + 2^{k-1})$$

$k = \frac{n-1}{2}$

$$= 2^{\frac{n-1}{2}} T(1) + c(2^k - 1)$$

$$= 2^{\frac{n-1}{2}} \times c_1 + c\, 2^{\frac{n-1}{2}} - c$$

$T(n) > c_2 2^{\frac{n-1}{2}} - c_3$

$O(2^{n/2})$

Intelligent guesses

- Guess a solution for the recurrence
- Plug the solution into the recurrence relation
- If a solution exists or we can prove the inequality, then our guess is correct

Fibonacci numbers

$$f(n) = f(n-1) + f(n-2)$$
$$f(0) = 0, f(1) = 1$$

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

```
f(2) = 1
f(3) = 2
f(4) = 3
f(5) = 5
f(6) = 8
f(7) = 13
f(8) = 21
f(9) = 34
f(10) = 55
f(11) = 89
f(12) = 144
..
```

Fibonacci numbers

$$f(n) = f(n-1) + f(n-2)$$
$$f(0) = 0, f(1) = 1$$

```
f(2) = 1
f(3) = 2
f(4) = 3
f(5) = 5
f(6) = 8
f(7) = 13
f(8) = 21
f(9) = 34
f(10) = 55
f(11) = 89
f(12) = 144
..
```

guess: $a2^n$

.5 1.5 2 3.5 5.5...

Fibonacci numbers

$$f(n) = f(n-1) + f(n-2)$$
$$f(0) = 0, f(1) = 1$$

$$S1 = c\left(\frac{1+\sqrt{5}}{2}\right)^n$$

$$S2 = c\left(\frac{1-\sqrt{5}}{2}\right)^n$$

```
f(2) = 1
f(3) = 2
f(4) = 3
f(5) = 5
f(6) = 8
f(7) = 13
f(8) = 21
f(9) = 34
f(10) = 55
f(11) = 89
f(12) = 144
..
```

guess: cx^n

$$cx^n$$

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

$$x^2 = x + 1$$

$$x^2 - x - 1 = 0$$

$$\alpha = \frac{1 + \sqrt{1 - 4 \times 1 \times (-1)}}{2 \times 1} = \frac{1+\sqrt{5}}{2}$$

$$\beta = \frac{1-\sqrt{5}}{2}$$

Fibonacci numbers

$$f(n) = f(n-1) + f(n-2)$$

$$f(0) = 0 \qquad f(1) = 1$$

$$S1 = c\left(\frac{1+\sqrt{5}}{2}\right)^n \qquad S2 = c\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$$A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n \text{ is also a solution.} \quad -①$$

$$A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n = A\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} + B\left(\frac{1-\sqrt{5}}{2}\right)^{n-1} + A\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} + B\left(\frac{1-\sqrt{5}}{2}\right)^{n-2}$$

S1: $c\left(\frac{1+\sqrt{5}}{2}\right)^n = c\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} + c\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} \Rightarrow A\left(\frac{1+\sqrt{5}}{2}\right)^n = A\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} + B\left(\frac{1+\sqrt{5}}{2}\right)^{n-1}$ — ②

S2: $c\left(\frac{1-\sqrt{5}}{2}\right)^n = c\left(\frac{1-\sqrt{5}}{2}\right)^{n-1} + c\left(\frac{1-\sqrt{5}}{2}\right)^{n-1}$

$B\left(\frac{1-\sqrt{5}}{2}\right)^n = B\left(\frac{1-\sqrt{5}}{2}\right)^{n-1} + B\left(\frac{1-\sqrt{5}}{2}\right)^{n-1}$ — ③

Fibonacci numbers

$$f(n) = f(n-1) + f(n-2)$$

$$\mathrm{c}x^n = cx^{n-1} + cx^{n-2}$$
$$x^2 = x + 1$$
$$x^2 - x - 1 = 0$$

roots = $\frac{-(-1) \pm \sqrt{(-1)^2 - 4*1*(-1)}}{2*1} = \frac{1 \pm \sqrt{5}}{2}$

Fibonacci numbers

- The general solution is of the form
  - $\mathrm{A}\alpha^n + B\beta^n \quad where\ c\alpha^n\ and\ c\beta^n\ are\ solutions\ of\ the\ recurrence\ relation$

$f(n) = f(n-1) + f(n-2)$
prove,
$\mathrm{A}\alpha^n + B\beta^n = (A\alpha^{n-1} + B\beta^{n-1}) + (A\alpha^{n-2} + B\beta^{n-2}) \qquad \rightarrow \text{ eq1}$

We already know:
$c\alpha^n = c\alpha^{n-1} + c\alpha^{n-2} \quad \rightarrow \quad A\alpha^n = A\alpha^{n-1} + A\alpha^{n-2} \qquad \rightarrow eq2$
$c\beta^n = c\beta^{n-1} + c\beta^{n-2} \quad \rightarrow \quad B\beta^n = B\beta^{n-1} + B\beta^{n-2} \qquad \rightarrow eq3$

Adding eq2 and eq3 we get eq1

Fibonacci numbers

$F(0) = 0$
$F(1) = 1$

- The general solution for $f(n) = f(n-1) + f(n-2)\ is$:

$$A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$A + B = 0$
$\left(\frac{1+\sqrt{5}}{2}\right)A + \left(\frac{1-\sqrt{5}}{2}\right)B = 1$

Solving for $f(0) and\ f(1), we\ get$: $A = \frac{1}{\sqrt{5}},\ \ B = -\frac{1}{\sqrt{5}}$

Final solution: $\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$

Solving recurrence relation

- Expansion method
- Intelligent guesses
- **Elimination of history**
  - **Full-history recurrence relation**
    - **Depends on all preceding terms instead of a few**
      - **Will discuss later**

Merge sort

References

- Read section-2.3.1 from the Cormen, Leiserson, Rivest, Stein book

Merge sort

- Merge sort is a recursive algorithm

- Base condition:
  - if the input array has one or zero elements, the array is already sorted; return the array

- Recursive step
  - Divide the array into two sub-arrays, A_1 and A_2
    - A_1 contains the first half, A_2 contains the second half
  - Recursively sort A_1 and A_2
  - Merge sorted A_1 and A_2 into a single sorted array

Merge sort

D0	[17	3	13	49	29	6	61	23]
D0	[3	6	13	17	23	29	49	61]
D1	[17	3	13	49]	[29	6	61	23]
D1	[3	13	17	49]	[6	23	29	61]
D2	[17	3]	[13	49]	[29	6]	[61	23]
D2	[3	17]	[13	49]	[6	29]	[23	61]
D3	[17]	[3]	[13]	[49]	[29]	[6]	[61]	[23]
D3	[17]	[3]	[13]	[49]	[29]	[6]	[61]	[23]

Merge sort

arr = [3 13 17 49] [6 23 29 61]

tmp1 = [3 13 17 49 ∞] tmp2 = [6 23 29 61 ∞]

i=0 i=1 i=2 i=3 i=4

tmp1[i] < tmp2[j]

j=0 j=1 j=2 j=3

[3 6 13 17 23 29 49 61]

Merge sort

```
void mergesort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int mid = (lo + hi) / 2;
  // recursively sort the first half
  mergesort(arr, lo, mid);
  // recursively sort the second half
  mergesort(arr, mid+1, hi);
  // merge the sorted arrays
  merge(arr, lo, mid, hi);
}
```

$T(n) = 2T(n/2) + c_1 n + c_2$

```
void merge(int arr[], int lo, int mid, int hi)
{
  int i, j, k;
  int n_elem_1 = mid - lo + 1;
  int n_elem_2 = hi - mid;

  int tmp1[n_elem_1 + 1];
  int tmp2[n_elem_2 + 1];

  for (i = 0; i < n_elem_1; i++) {
    tmp1[i] = arr[lo + i];
  }
  for (j = 0; j < n_elem_2; j++) {
    tmp2[j] = arr[mid + 1 + j];
  }
  tmp1[i] = tmp2[j] = INT_MAX;
  i = j = 0;

  for (k = lo; k <= hi; k++) {
    if (tmp1[i] <= tmp2[j]) {
      arr[k] = tmp1[i];  i += 1;
    } else {
      arr[k] = tmp2[j];  j += 1;
    }
  }
}
```

Merge sort

D0	[17	3	13	49	29	6	61	23]
D0	[3	6	13	17	23	29	49	61]
D1	[17	3	13	49]	[29	6	61	23]
D1	[3	13	17	49]	[6	23	29	61]
D2	[17	3]	[13	49]	[29	6]	[61	23]
D2	[3	17]	[13	49]	[6	29]	[23	61]
D3	[17]	[3]	[13]	[49]	[29]	[6]	[61]	[23]
D3	[17]	[3]	[13]	[49]	[29]	[6]	[61]	[23]

Merge sort

- Time complexity

$T(1) = c$

$T(n) = 2T(n/2) + c_1 n + c_2$

$T(n/2) = 2T\left(\frac{n}{2^2}\right) + c_1\left(\frac{n}{2}\right) + c_2$

$T(n) = 2\left(2T\left(\frac{n}{2^2}\right) + c_1\left(\frac{n}{2}\right) + c_2\right) + c_1 n + c_2$

$= 2^2 T\left(\frac{n}{2^2}\right) + c_1 n + 2c_2 + c_1 n + c_2$

$= 2^2 T\left(\frac{n}{2^2}\right) + 2c_1 n + 2c_2 + c_2$

$= 2^2\left(2T\left(\frac{n}{2^3}\right) + c_1\left(\frac{n}{2^2}\right) + c_2\right) + 2c_1 n + 2c_2 + c_2$

$= 2^3 T\left(\frac{n}{2^3}\right) + 3c_1 n + 2^2 c_2 + 2c_2 + c_2$

$\cdots$

$= 2^k T\left(\frac{n}{2^k}\right) + kc_1 n + c_2(1 + 2 + 2^2 + \cdots + 2^{k-1})$

Substitute $n = 2^k$

$= nT(1) + c_1 n \log_2 n + c_2(2^k - 1)$

$= nc + c_1 n \log_2 n + c_2 n - c_2$

$O(n \log n)$

Merge sort

$$T(1) = 2$$

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + c_1 n + c_2 \\
&= 2\left(2T\left(\frac{n}{2^2}\right) + c_1\left(\frac{n}{2}\right) + c_2\right) + c_1 n + c_2 \\
&= 2^2 T\left(\frac{n}{2^2}\right) + 2c_1 n + 2c_2 + c_2 \\
&= 2^2\left(2T\left(\frac{n}{2^3}\right) + c_1\left(\frac{n}{2^2}\right) + c_2\right) + 2c_2 + c_2 \\
&= 2^3 T\left(\frac{n}{2^3}\right) + 3c_1 n + 2^2 c_2 + 2c_2 + c_2 \\
&= \ldots \\
&= 2^k T\left(\frac{n}{2^k}\right) + kc_1 n + c_2\left(1 + 2 + 2^2 + \cdots + 2^{k-1}\right)
\end{aligned}$$

Substitute $n = 2^k$

$$\begin{aligned}
T(n) &= nT(1) + c_1 nlogn + c_2\left(2^k - 1\right) \\
&= 2n + c_1 nlogn + c_2(n-1) \ = O(nlogn)
\end{aligned}$$

Quick sort

References

- Read section-6.4.4 from the Udi Manber book

Quick sort

- Quick sort is a recursive algorithm

- Base condition:
  - if the input array has one or zero elements, the array is already sorted; return the array

- Recursive step
  - Select an element `x` as pivot
  - Partition the array `A[lo .. hi]` into two sub-arrays `A[lo .. p-1]` and `A[p+1 .. hi]`
    - Every element in `A[lo .. p-1]` $\leq$ `A[p]`
    - Every element in `A[p+1 ]` $\geq$ `A[p]`
    - `A[p] = x  // so x is at its proper position in the array`

  - Recursively sort `A[lo .. p-1]` and `A[p+1 .. hi]`
  - No need to combine the results!

Partition

	47	21	3	83	17	59	43	81	11	99
PART	[21	3	17	43	11]	47	[83	81	59	99]
SORT	[21	3	17	43	11]		[83	81	59	99]

	47	21	3	83	17	59	43	81	11	99
PART	17	21	11	3	43	47	81	59	99	83
SORT										

The first element of the array is the pivot. You can partition the array in multiple ways as long as the post-conditions for partitioning are met.

In-class quiz in the next class

(Syllabus: lecture 1 to 5)

Closed book exam

Ungraded homework

- Write a program to print all permutations (${}^{n}P_{n}$) of an n element array
- Compute the time complexity of your permutation algorithm
- In the merge routine of the merge-sort algorithm, we create copies of both left and right (sorted) sub-arrays before merging them. However, when we update the original right sub-array in the algorithm, the previous value of the updated element is not required after the update. Can you rewrite the merge routine such that it doesn't make a copy of the right (sorted) subarray?
- https://github.com/Systems-IIITD/DSA_Lab
  - Please refer to the README file

Today’s topics

- Quick sort
- Pointers

Quick sort

References

- Read section-6.4.4 from the Udi Manber book

Quick sort

- Quick sort is a recursive algorithm

- Base condition:
  - if the input array has one or zero elements, the array is already sorted; return the array

- Recursive step
  - Select an element `x` as pivot
  - Partition the array `A[lo .. hi]` into two sub-arrays `A[lo .. p-1]` and `A[p+1 .. hi]`
    - Every element in `A[lo .. p-1]` $\leq$ `A[p]`
    - Every element in `A[p+1 ]` $\geq$ `A[p]`
    - `A[p] = x  // so x is at its proper position in the array`

  - Recursively sort `A[lo .. p-1]` and `A[p+1 .. hi]`
  - No need to combine the results!

Partition

						i				
	47	21	3	83	17	59	43	81	11	99
PART	[21	3	17	43	11]	47	[83	81	59	99]
SORT	[21	3	17	43	11]		[83	81	59	99]

	47	21	3	83	17	59	43	81	11	99
PART	17	21	11	3	43	47	81	59	99	83
SORT										

The first element of the array is the pivot. You can partition the array in multiple ways as long as the post-conditions for partitioning are met.

Partition

- `pivot = arr[0]`
- `Set left = 0, right = n-1`
- `while (arr[left] <= pivot) left += 1`
- `while (arr[right] > pivot) right -= 1`
- `if (left < right) exchange arr[right] with arr[left]`
- `if (right < left) exchange arr[0]` (pivot) `with arr[right]`

	47	21	3	83	17	59	43	81	11	99
PART	47	21	3	11.	17	59	43	81	83	99
	47	21	3	11	17	43	59	81	83	99
	43	21	3	11	17	$\boxed{47}$	59	81	83	99

Partition

- `pivot = arr[0]`
- `Set left = 0, right = n-1`
- `while (arr[left] <= pivot) left += 1`
- `while (arr[right] > pivot) right -= 1`
- `if (left < right) exchange arr[right] with arr[left]`
- `if (right < left) exchange arr[0] (pivot) with arr[right]`

	47	21	3	83	17	59	43	81	11	99
PART	47	21	3	**11**	17	59	43	81	**83**	99
	47	21	3	11	17	**43**	**59**	81	83	99
	43	21	3	11	17	**47**	59	81	83	99

Partition and recur

D0	47	21	3	83	17	59	43	81	11	99
P	47	21	3	**11**	17	59	43	81	**83**	99
	47	21	3	11	17	**43**	**59**	81	83	99
	43	21	3	11	17	**47**	59	81	83	99
D1	43	21	3	11	17		59	81	83	99
P	17	21	3	11	43		59	81	83	99
D2	17	21	3	11				81	83	99
P	17	11	3	21				81	83	99
	3	11	17	21						
D3	3	11		21					83	99
P	3	11							83	99
D4		11								99

Partition and recur

D0	47	21	3	83	17	59	43	81	11	99
P	47	21	3	**11**	17	59	43	81	**83**	99
	47	21	3	11	17	**43**	**59**	81	83	99
	43	21	3	11	17	**47**	59	81	83	99
D1	43	21	3	11	17		59	81	83	99
P	**17**	21	3	11	**43**		**59**	81	83	99
D2	17	21	3	11				81	83	99
P	17	**11**	3	**21**				**81**	83	99
	3	11	**17**	21						
D3	3	11		21					83	99
P	**3**	11		**21**					**83**	99
D4		**11**								**99**

Quick sort

```
void quicksort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int p = partition(arr, lo, hi);
  quicksort(arr, lo, p - 1);
  quicksort(arr, p + 1, hi);
}
```

```
int partition(int arr[], int lo, int hi)
{
  int pivot = arr[lo];
  int left = lo;
  int right = hi;

  while (left < right) {
    while (left <= right && arr[left] <= pivot) {
      left += 1;
    }
    while (right >= left && arr[right] > pivot) {
      right -= 1;
    }
    if (left < right) {
      exchange(arr, left, right);
    }
  }

  exchange(arr, lo, right);
  return right;
}
```

Worst case partition

- pivot = arr[0]
- Set left = 0, right = n-1
- while (arr[left] <= pivot) left += 1
- while (arr[right] > pivot) right -= 1
- if (left < right) exchange arr[right] with arr[left]
- if (right < left) exchange arr[0] (pivot) with arr[right]

In the worst case, the partition algorithm generates two arrays of size zero and (n-1) for further sorting.

	11	21	31	33	37	49	53	61	73	93
PART	11	21	31	33	37	49	53	61	73	93
D1										

Time complexity

```
void quicksort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int p = partition(arr, lo, hi);
  quicksort(arr, lo, p - 1);
  quicksort(arr, p + 1, hi);
}
```

Worst case:

$T(n) = T(n-1) + c_1 \times n + c_2$

$O(n^2)$

```
int partition(int arr[], int lo, int hi)
{
  int pivot = arr[lo];
  int left = lo;
  int right = hi;

  while (left < right) {
    while (left <= right && arr[left] <= pivot) {
      left += 1;
    }
    while (right >= left && arr[right] > pivot) {
      right -= 1;
    }
    if (left < right) {
      exchange(arr, left, right);
    }
  }

  exchange(arr, lo, right);
  return right;
}
```

Time complexity

```
void quicksort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int p = partition(arr, lo, hi);
  quicksort(arr, lo, p - 1);
  quicksort(arr, p + 1, hi);
}
```

Best case:

$T(n) = 2T(n/2) + c_1 n + c_2$

$O(n \log n)$

```
int partition(int arr[], int lo, int hi)
{
  int pivot = arr[lo];
  int left = lo;
  int right = hi;

  while (left < right) {
    while (left <= right && arr[left] <= pivot) {
      left += 1;
    }
    while (right >= left && arr[right] > pivot) {
      right -= 1;
    }
    if (left < right) {
      exchange(arr, left, right);
    }
  }

  exchange(arr, lo, right);
  return right;
}
```

Time complexity

```
void quicksort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int p = partition(arr, lo, hi);
  quicksort(arr, lo, p - 1);
  quicksort(arr, p + 1, hi);
}
```

Average case:

$$T(n) = T(i) + T(n-1-i) + c_1 n + c_2$$

$$= \left(\frac{1}{n}\sum_{i=0}^{n-1}\left(T(i) + T(n-1-i)\right)\right) + c_1 n + c_2$$

```
int partition(int arr[], int lo, int hi)
{
  int pivot = arr[lo];
  int left = lo;
  int right = hi;

  while (left < right) {
    while (left <= right && arr[left] <= pivot) {
      left += 1;
    }
    while (right >= left && arr[right] > pivot) {
      right -= 1;
    }
    if (left < right) {
      exchange(arr, left, right);
    }
  }

  exchange(arr, lo, right);
  return right;
}
```

Time complexity (average case)

$$T(n) = \left(\frac{1}{n}\left(\sum_{i=0}^{n-1}(T(i)+T(n-1-i))\right)\right) + c_1 n + c_2$$

$$= \frac{1}{n}\left(\sum_{i=0}^{n-1}T(i) + \sum_{i=n-1}^{0}T(i)\right) + c_1 n + c_2$$

$$= \frac{2}{n}\sum_{i=0}^{n-1}T(i) + c_1 n + c_2$$

$$nT(n) = 2\sum_{i=0}^{n-1}T(i) + c_1 n^2 + c_2 n \quad \text{— (1)}$$

$$(n+1)T(n+1) = 2\sum_{i=0}^{n}T(i) + c_1(n+1)^2 + c_2(n+1) \quad \text{— (2)}$$

$$(n+1)T(n+1) - nT(n) = 2T(n) + c_1(n+1)^2 + c_2(n+1) - c_1 n^2 - c_2 n$$

$$(n+1)T(n+1) = T(n)(n+2) + c_1(2n+1) + c_2$$

$$T(n+1) = \frac{n+2}{n+1}T(n) + \frac{c_1(2n+1)}{n+1} + \frac{c_2}{n+1}$$

$$= \frac{n+2}{n+1}T(n) + 2 + \frac{c_2 - c_1}{n+1} \leq \frac{n+2}{n+1}T(n) + c_3$$

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

Time complexity (average case)

$$T(n+1) \le \frac{n+2}{n+1} T(n) + c_3$$

$$\underline{T(n)} \le \frac{n+1}{n} T(n-1) + c_3$$

$$= \frac{n+1}{n}\left[\frac{n}{n-1} T(n-2) + c_3\right] + c_3 = \frac{n+1}{n-1} \underline{T(n-2)} + c_3 \frac{n+1}{n} + c_3$$

$$= \frac{n+1}{n-1}\left(\frac{n-1}{n-2} T(n-3) + c_3\right] + c_3 \frac{n+1}{n} + c_3$$

$$= \frac{n+1}{n-2} T(n-3) + c_3 \frac{n+1}{n-1} + c_3 \frac{n+1}{n} + c_3$$

$$\cdots$$

$$= \frac{n+1}{n-k+1} T(n-k) + c_3\left(1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \cdots + \frac{n+1}{n-k+2}\right)$$

Substitute, $k = n-1$

$$T(n) = \frac{n+1}{2} T(1) + c_3 (n+1)\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{3}\right)$$

Time complexity (average case)

$$T(n) = \frac{n+1}{2} T(1) + c_3 (n+1) \left(H(n+1) - \frac{3}{2} \right)$$

$$= \frac{n+1}{2} c + c_3 (n+1) \left(O(\log(n+1)) - \frac{3}{2} \right)$$

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

Harmonic series sum

- https://en.wikipedia.org/wiki/Harmonic_series_(mathematics)
  - Comparison test
  - Integral test

Harmonic series sum

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$$

$$\leq 1 + \left(\underline{\frac{1}{2} + \frac{1}{2}}\right) + \left(\underline{\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}}\right) +$$

$$\leq K$$

$$2^0 + 2^1 + 2^2 + \cdots + 2^{K-1} = n$$

$$2^K - 1 = n$$

$$K = \log(n+1)$$

$$H(n) \leq \log(n+1)$$

Harmonic series sum

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

$$H(n) \geq 1 + \frac{1}{2} + \left(\frac{1}{4} + \frac{1}{4}\right) + \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right) + \cdots$$

$$\geq 1 + \frac{k}{2}$$

$$2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{k-1} = n - 1$$

$$2^k - 1 = n - 1$$

$$k = \log_2 n$$

$$H(n) \geq 1 + \frac{\log_2 n}{2}$$

$$1 + \frac{\log_2 n}{2} \leq H(n) \leq \log_2 (n+1)$$

Time complexity (average case)

- Time complexity, when the portioning algorithm returns i

$$T(n) = T(i) + T(n-1-i) + c_1 n + c_2$$

If each index has the same probability of being selected as a target position for the pivot, then the average complexity is

$$\mathrm{T(n)} = \frac{1}{n}\left(\sum_{i=0}^{n-1}(T(i) + T(n-1-i))\right) + \mathrm{c_1 n} + \mathrm{c_2}$$

Time complexity (average case)

$$\mathrm{T(n)} = \frac{1}{n}\left(\sum_{i=0}^{n-1}(T(i) + T(n-1-i))\right) + (\mathrm{c_1 n} + \mathrm{c_2})$$

$$\mathrm{T(n)} = \frac{2}{n}\left(\sum_{i=0}^{n-1}(T(i)\right) + (\mathrm{c_1 n} + \mathrm{c_2})$$

(Full history recurrence relation)

$$\mathrm{nT(n)} = 2\left(\sum_{i=0}^{n-1}(T(i)\right) + \mathrm{c_1 n^2} + \mathrm{c_2 n}$$

$$(\mathrm{n}+1)\mathrm{T(n}+1) = 2\left(\sum_{i=0}^{n}(T(i)\right) + \mathrm{c_1}(\mathrm{n}+1)^2 + \mathrm{c_2}(\mathrm{n}+1)$$

$$(\mathrm{n}+1)T(n+1) - nT(n) = 2T(n) + c_1(2n+1) + c_2$$

$$T(n+1) = \frac{n+2}{n+1}T(n) + \frac{c_1(2n+1)}{n+1} + \frac{c_2}{n+1} \leq \frac{n+2}{n+1}T(n) + c$$

Time complexity(average case)

$$T(n+1) \leq \frac{n+2}{n+1} T(n) + c$$

$$T(n) \leq \frac{n+1}{n} T(n-1) + c$$

$$= \frac{n+1}{n}\left(\frac{n}{n-1} T(n-2) + c\right) + c \ = \frac{n+1}{n-1} T(n \ -2) + c\left(1 + \frac{n+1}{n}\right)$$

$$= \frac{n+1}{n-1}\left(\frac{n-1}{n-2} T(n-3) + c\right) + c\left(1 + \frac{n+1}{n}\right) = \frac{n+1}{n-2} T(n-3) + c\left(1 + \frac{n+1}{n} + \frac{n+1}{n-1}\right)$$

$$= \ \ldots$$

$$= \frac{n+1}{n-k+1} T(n-k) + c\ \left(1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \ \ldots + \frac{n+1}{n-k+2}\right)$$

Time complexity (average case)

Substituting k = n – 1

$$T(n) \leq \frac{n+1}{2} T(1) + c\ \left(1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \ \ldots + \frac{n+1}{3}\right)$$

$$= \frac{n+1}{2} c_3 + c(n+1)\left(1 + \frac{1}{2} + \frac{1}{3} + .. + \frac{1}{n+1}\ - \frac{3}{2}\right)$$

$$Substituting, 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\ with\ O(\log n)$$

$$= \frac{n+1}{2} c_3 + c(n+1)\ (O(\log(n+1)) - \frac{3}{2})\ = O(n \log n)$$

Harmonic series sum

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

$$\leq 1 + \left(\frac{1}{2} + \frac{1}{2}\right) + \left(\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}\right) + \cdots$$

$$\leq 1 + 1 + 1 + \ \ldots upto\ k\ terms\ \ =\ \ \mathrm{k}$$

$$1 + 2 + 2^2 + \cdots + 2^{k-1} = n$$
$$2^k - 1 = n$$
$$k = \log_2(n+1)$$
$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \leq \log_2(n+1)$$

Harmonic series sum

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

$$\geq 1 + \frac{1}{2} + \left(\frac{1}{4} + \frac{1}{4}\right) + \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right) + \cdots$$

$$\geq 1 + \frac{k}{2}$$

$$1 + 2 + 2^2 + \cdots + 2^{k-1} = n - 1$$
$$2^k - 1 = n - 1$$
$$k = \log_2 n$$
$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \geq 1 + \frac{\log_2 n}{2}$$

Quick sort vs merge sort

Quick sort

- The worst-case time complexity is O(n^2)
- Require very little extra space (an in-place algorithm)
  - Good cache locality
- Empirically, quick sort performs better than merge sort

Merge sort

- The worst-case time complexity is O($n \log n$)
- Requires extra space during the merge operation (not in-place)
  - Poor cache locality
- Empirically, merge sort performs poorly than quick sort

Quick sort

- C++ standard library implements a hybrid sorting algorithm, Intro-sort, to mitigate the worst-case behavior of the quick sort
  - https://en.wikipedia.org/wiki/Introsort
  - Quick sort + Heap sort (worst case $n \log n$) + Insertion sort ($O(n^2)$)

- Other techniques to reduce the probability of worst-case behaviour
  - Select the median of the first, middle, and last element as the pivot
  - Randomly select the pivot

Memory layout

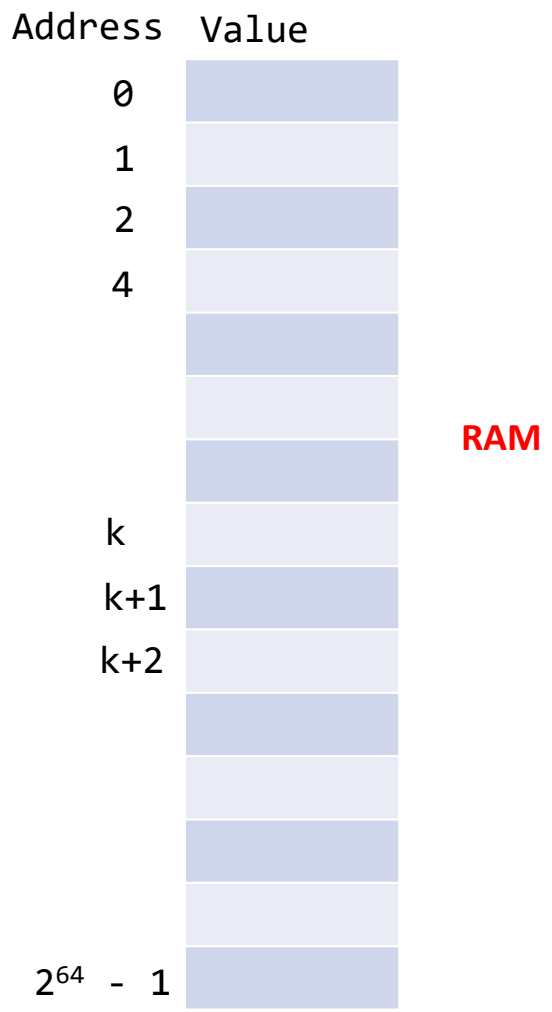
Every byte of the RAM has a unique address.

For every variable in your program, the compiler allocates space of size "sizeof(typeof(variable))" in the RAM.

The address of (&) operator returns the address of a variable.

int a;

&a



Pointers

- A pointer is a variable that contains the address of a variable or a RAM address

```
p int a = 10;
q int *x = &a;     // int*   (address of a variable of type int)
r int **y = &x;    // int**  (address of a variable of type int*)
s int ***z = &y;   // int*** (address of a variable of type int**)
```

Today’s topics

- Pointers
- Structures
- Quiz

Pointers

Memory layout

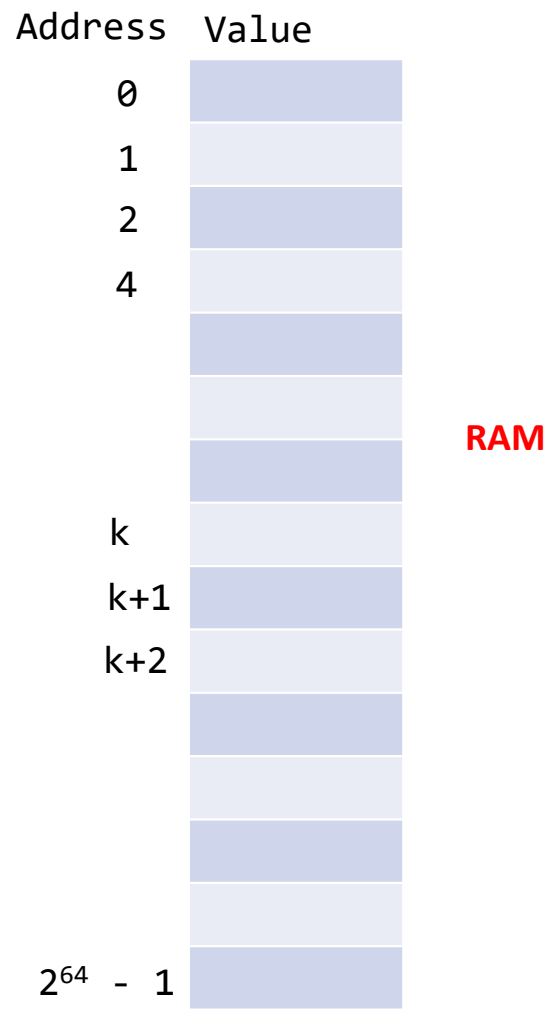
Every byte of the RAM has a unique address.

For every variable in your program, the compiler allocates space of size "sizeof(typeof(variable))" in the RAM.

The address of (&) operator returns the address of a variable.

int a;

&a



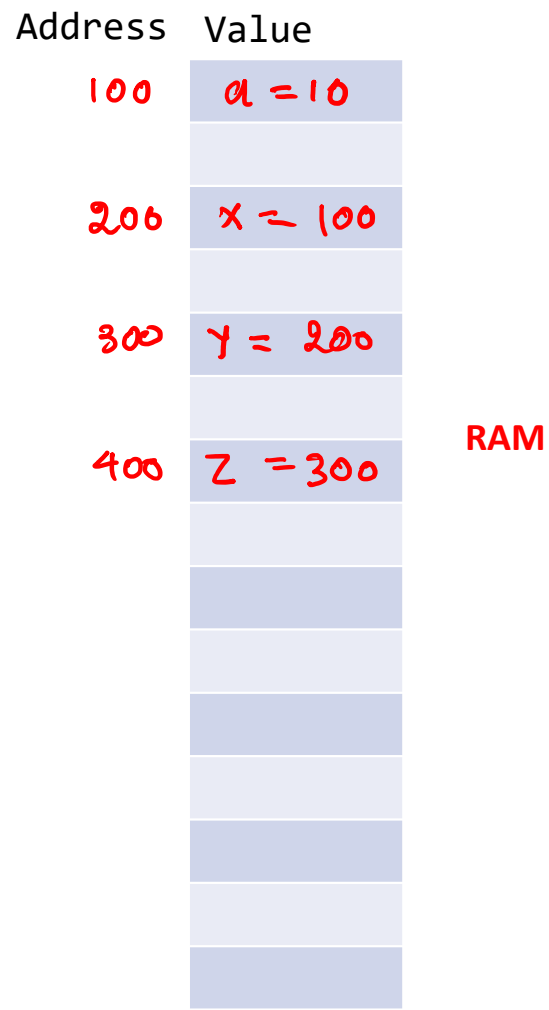
Pointers

- A pointer is a variable that contains the address of a variable or a RAM address

```
int a = 10;
// application can’t guess &a
int *x = &a;    // int*   (address of a variable of type int)
// application can’t guess &x
int **y = &x;   // int**  (address of a variable of type int*)
// application can’t guess &y
int ***z = &y;  // int*** (address of a variable of type int**)
// application can’t guess &z
```

Pointers

```
int a = 10;
int *x = &a;
int **y = &x;
int ***z = &y
```

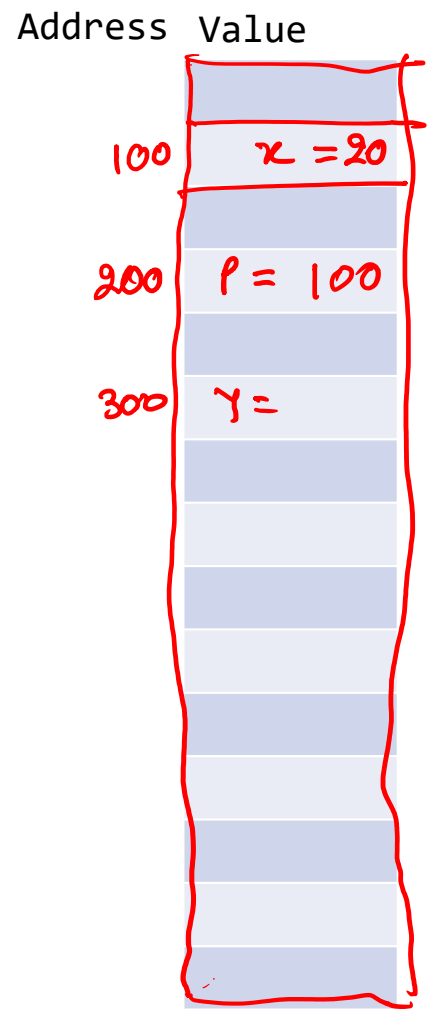


Can we guess these addresses at compile time?

Reading from a memory address

- You can read the value(s) of some memory location(s) using a pointer

```
int x = 20;
int *p = &x;
int y;

y = *p;   20

How many bytes are read?  // sizeof(typeof(*p))
What is the value of y?   20
```



Reading from a memory address

- You can read the value(s) of some memory location(s) using a pointer

```
int x = 20;
int *p = &x;
int **q = &p;
int *y;

y = *q;

How many bytes are read?  // sizeof(typeof(*q)) 8
What is the value of y?   100

int z = *y;
How many bytes are read? 4
What is the value of z? 25
```

Address	Value
100	x=20
200	p = 100
300	q = 200
400	y = 100

Reading from a memory address

- You can read the value(s) of some memory location(s) using a pointer

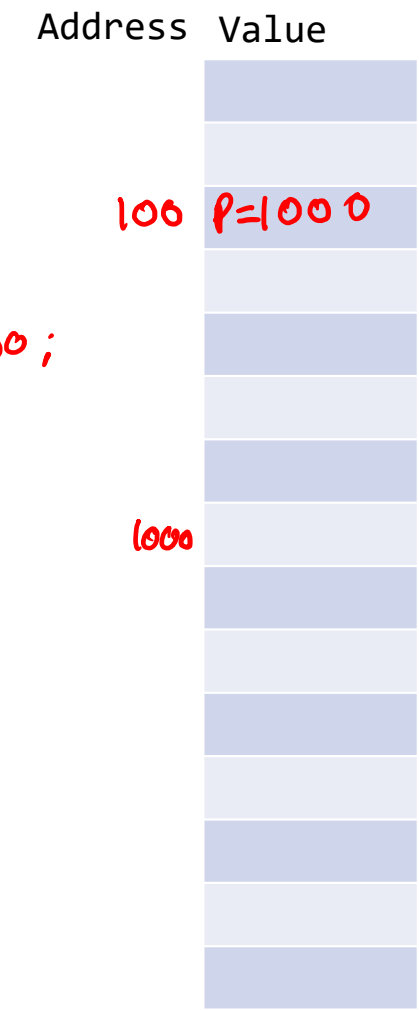
```
int *p = (int*)1000;
int y;

y = *p;
```

char *c = (char*)1000;
char c1 = *c;

How many bytes are read?

What are the memory locations whose values are read during the memory read?

How can we read just one byte? 4



Writing to a memory address

- You can update the value(s) of some memory location(s) using a pointer

```
int x;
int *p = &x;

*p = 1;

How many bytes are written? sizeof(typeof(*p))  4
What is the value of x?  1
What is the value of p?  100
```

Address	Value
100	x = 1
200	P= 100

Writing to a memory address

- You can update the value(s) of some memory location(s) using a pointer

```
int x;
int *p = &x;
int **q = &p;

*q = 1;
```

How many bytes are written? sizeof(typeof(*q)) 8

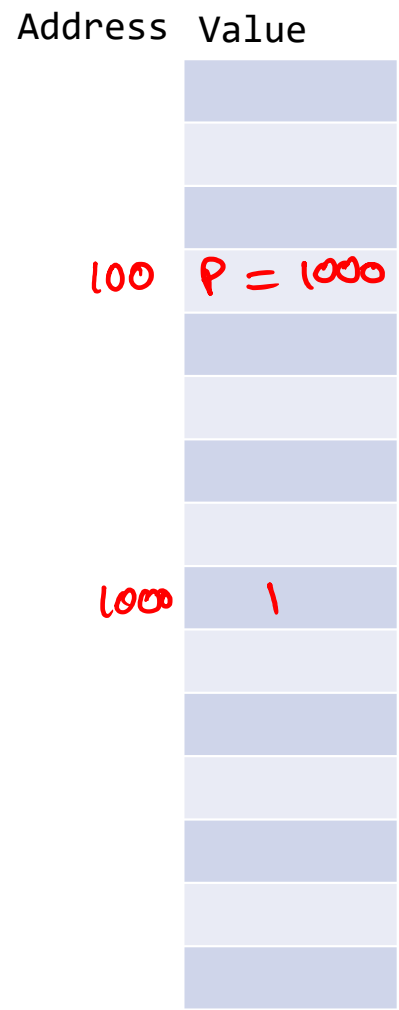
What is the value of x? garbage
What is the value of p? 1
What is the value of q? 200

Address	Value
100	X
200	P = ~~100~~ 1
300	q = 200

Writing to a memory address

- You can update the value(s) of some memory location(s) using a pointer

```
int *p = (int*)1000;
*p = 1;

How many bytes are written?

What are the memory locations whose values are
updated during the memory write?

How can we update just one byte?
```



Pointers

```
int a = 10;
printf("%p %d", &a, a);
int *p1 = &a;
printf("%p %p %d", p1, &p1, *p1);
int *p2 = p1;
printf("%p %p %d", p2, &p2, *p2);
int **p3 = &p1;
printf("%p %p %d", p3, *p3, **p3);
*p1 = 20;
printf("%p %d %p %d", p1, *p1, p2, *p2);
*p2 = 30;
printf("%p %d %p %d", p1, *p1, p2, *p2);
*p3 = (int*)40;
printf("%p %p %p %d", p3, *p3, p1, *p1);
*p1 = 50;
printf("%p %d %p %d", p1, *p1, p2, *p2);
```

100 10

100 200 10

100 300 10

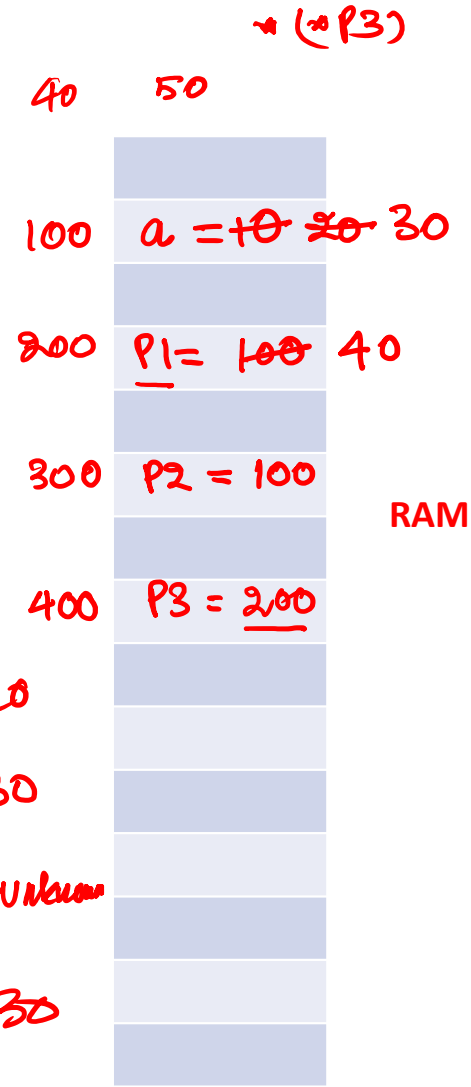
200 100 10

100 20 100 20

100 30 100 30

200 40 40 Unknown

40 50 100 30



Quiz (25 points)

1. Solve the following recurrence relation using the expansion method. [10]

$$T(n) = T(n-1) + c_1 n^2 + c_2$$

$$T(1) = c$$

2. Write an algorithm for the tower of Hanoi problem with **four** towers. In addition to the source and destination tower, there are two temporary towers. The goal is to move $\boldsymbol{n}$ discs from the source tower to the destination tower using temporary towers. For all $\boldsymbol{n \geq 4}$, your algorithm should take less number of moves than the number of moves in the three towers solution. [15]

Today’s topics

- Pointers
- Structures
- Linked lists

Pointers

Pointer arithmetic

- The rules for pointer arithmetic are different from integer arithmetic

```
int *p;

Let us say, at some point the value of p is 100.

p + 1 == 100 + 1×4 = 104
p + i == 100 + i×4

Syntactic sugar for (p + i) :   & p[i]
```

Pointer arithmetic

- The rules for pointer arithmetic are different from integer arithmetic

```
T *p;     // type of variable is T*
          // T is some type, e.g., int, int*, int**, char, etc.

Let us say, at some point the value of p is 100.

p + 1 == 100 + 1 * sizeof(T)
p + i ==  100 + i * sizeof(T)

Syntactic sugar for (p + i) :
```

Array

- The array elements are stored in contiguous memory locations

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

100 104 108 . . . .

```
int a[10];

The base address of the array "a" is &a[0].
Let us say the base address of the array is X (0 ≤ X ≤ 2^64 − 1).
In this case,
&a[1] == X + 4
&a[2] == X + 8
&a[3] == X + 12
…
&a[i]  == X + ?
```

Array

- The array elements are stored in contiguous memory locations

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

```
T a[10];      // T is some type, e.g., int, float, char, int*, etc.
```

The base address of the array "a" is &a[0].
Let us say the base address of the array is X ($0 \leq X \leq 2^{64} - 1$).
In this case,

```
&a[1] == X + sizeof(T) x 1
&a[2] == X + sizeof(T) x 2
&a[3] == X +
…
&a[i]  == X + sizeof(T) x i
```

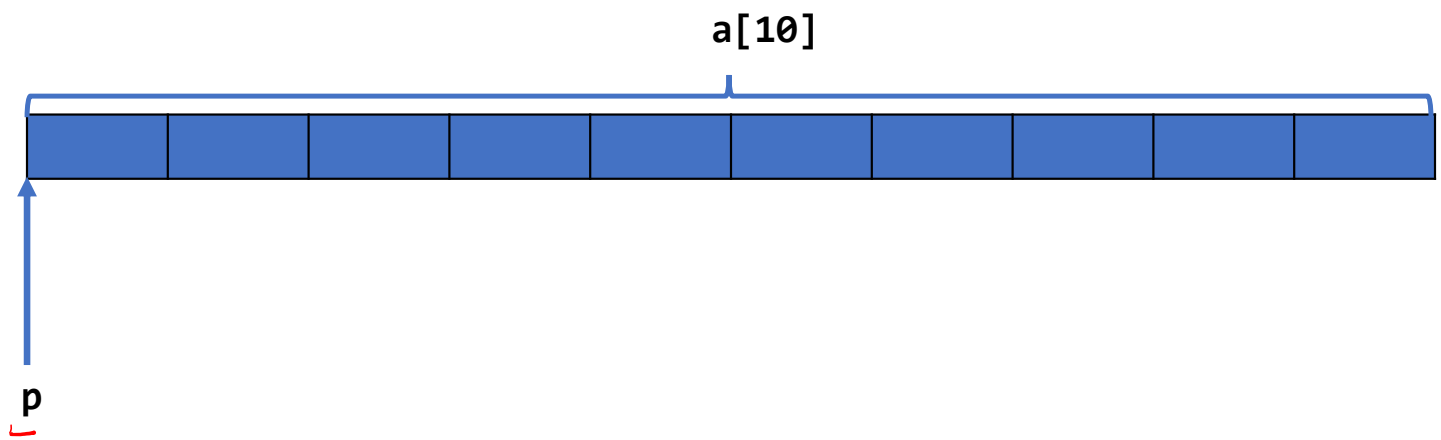
Array

- We can store the address of an array element in a pointer

```
int a[100];
int *p1, *p2;

p1 = &a[0];       // storing the base address of the array in p1
p2 = a;           // assignment of an array name to a pointer variable is allowed

p1 == p2          // value of a is same as &a[0]
```

Array

```
int a[10];
int *p = a;
```



Array

- We can store the address of an array element in a pointer
- Just like pointers, there are multiple syntaxes to access array elements

```
int a[100];
int *p1, *p2;

p1 = &a[0];       // storing the base address of the array in p1
p2 = a;           // assignment of an array name to a pointer variable is allowed

p1 == p2          // value of a is same as &a[0]

a[i] = 100;       // writing to the ith element
*(a + i) = 100;   // *(a + i) is the same as a[i]

(a + i) is same as &a[i]
```

malloc

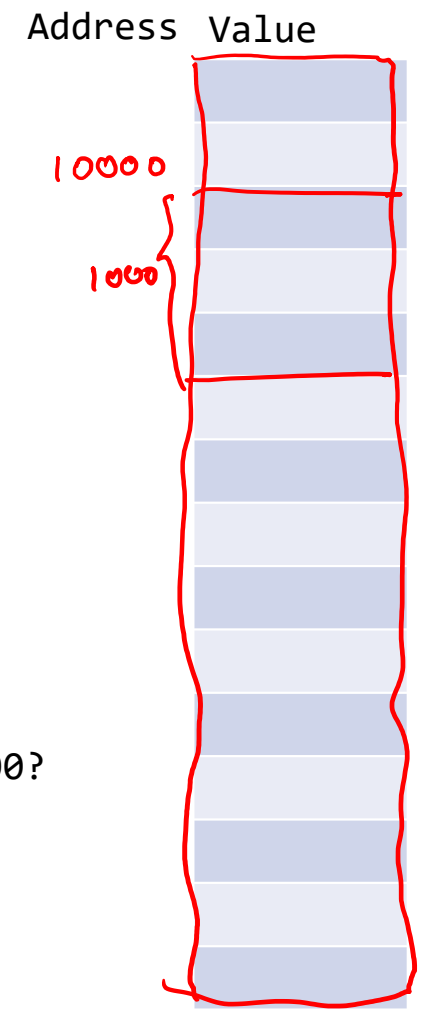
- At runtime, a program can allocate space from RAM using malloc

```
int *a = (int*)malloc(1000);

Application can't guess the return value of malloc.
Let us say the value of a is 10000.
Now, it is legal for the application to access
memory locations [10000, 10000+1000)

How can we update 4-bytes starting at memory location 10000 + 100?
```

~~*(a+100)~~

*(a+25)

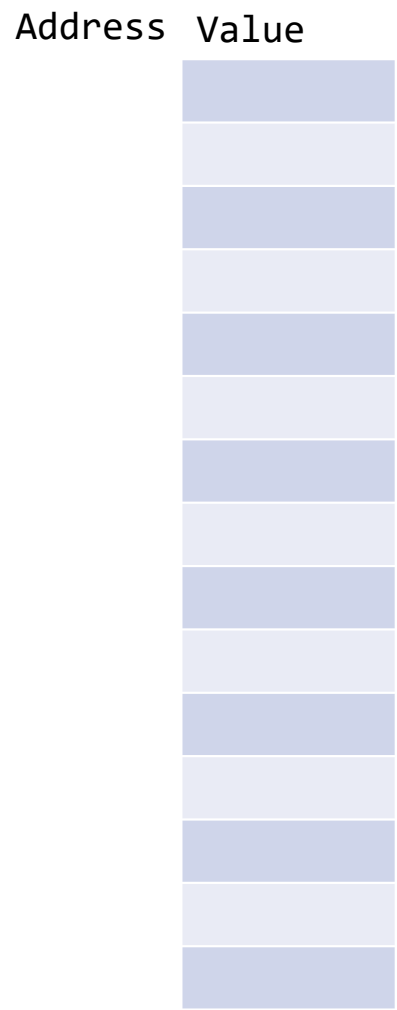
a[25]



malloc

- At runtime, a program can allocate space from RAM using malloc

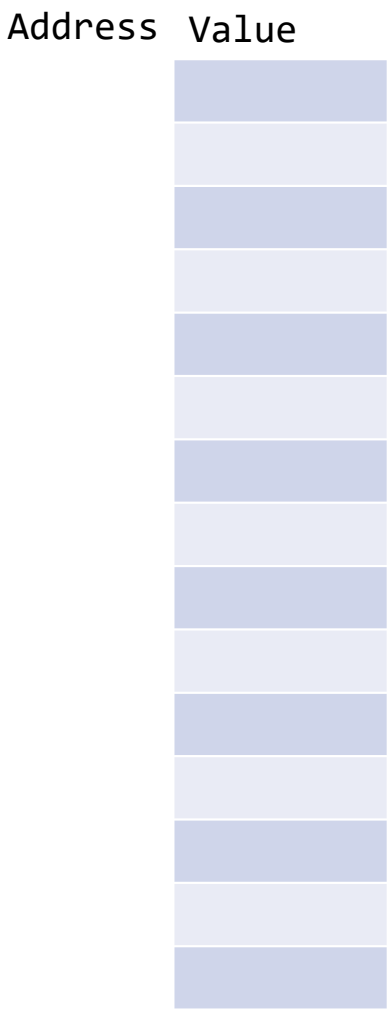
```
int *a = (int*)malloc(1000);

Application can't guess the return value of malloc.
Let us say the value of a is 10000.
Now, it is legal for the application to access
memory locations [10000, 10000+1000)


*a = 10;   // updating [10000, 10000+4)
*(a + 1) = 20; // updating [10000+4, 10000+8)
*(a + i) = i; // updating [10000+i*4, 10000+i*4+4)
```



malloc

- At runtime, a program can allocate space from RAM using malloc

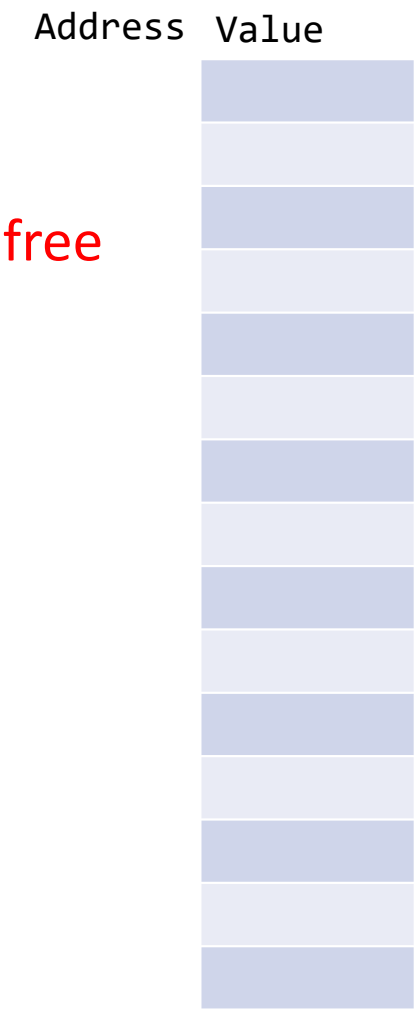
```
int *a = (int*)malloc(1000);

Application can't guess the return value of malloc.
Let us say the value of a is 10000.
Now, it is legal for the application to access
memory locations [10000, 10000+1000)


a[0] = 10;    // updating [10000, 10000+4)
a[1] = 20;    // updating [10000+4, 10000+8)
a[i] = i;     // updating [10000+i*4, 10000+i*4+4)

Syntactic sugar for *a => a[0]
                    *(a + i) => a[i]
```



free

- The malloced RAM addresses can be released using free

```
int *a = (int*)malloc(1000);

Application can’t guess the return value of malloc.
Let us say the value of a is 10000.
Now, it is legal for the application to access
memory locations [10000, 10000+1000)

/*
 * Some code here
 * /

free(a);
After free, it is illegal for the application to access
memory locations [10000, 10000+1000)
```

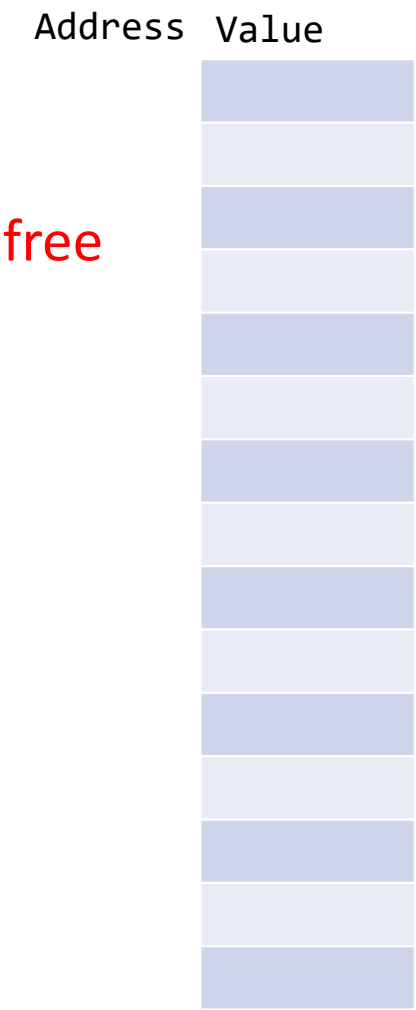


free

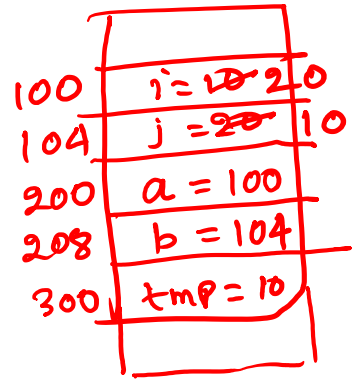
- The malloced RAM addresses can be released using free

What if an application never frees a malloced address?

memory leak



Passing pointers

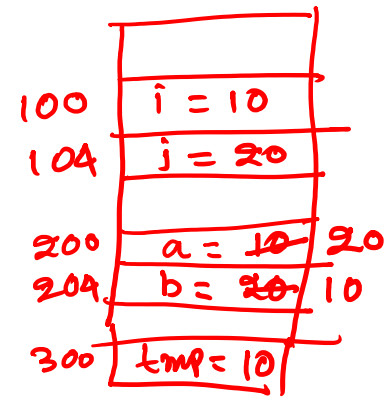



```
                    =100      =104
void swap(int *a, int *b) {
   int tmp = *a;  — 10
   *a = *b;
   *b = tmp;
}

int main() {
   int i = 10, j = 20;
   swap(&i, &j);
   printf("i=%d j=%d\n", i, j);   20  10
   return 0;
}
```

&i = 100
&j = 104

During function call:
int *a = &i
int *b = &j

Passing values




```
void swap(int a, int b) {
   int tmp = a;
   a = b;
   b = tmp;
}

int main() {
   int i = 10, j = 20;
   swap(i, j);
   printf("i=%d j=%d\n", i, j);
   return 0;
}
```

=i =j

10 20

During function call:
int a = i
int b = j

Array in function arguments

```
void foo(int arr[10]);

void foo(int arr[]);

void foo(int *arr);
```

All three declarations are the same.

Why does the C compiler ignore the number of elements in the first dimension in the array arguments?

```
void foo(int arr[10]) {
  …   arr[100] = 10;
}

int main() {
  int a[10];
  foo(a);
  return 0;
}
```

Array in function arguments

```
void foo(int arr[10]);

void foo(int arr[]);

void foo(int *arr);
```

All three declarations are the same.

Why does the C compiler ignore the number of elements in the first dimension in the array arguments?

```
void foo(int *arr) {
  arr[-1] = 1;
  arr[-2] = 2;
  …
}

int main() {
  int a[10];
  foo(&a[2]); // we can pass an internal
              // address of an array as
              // an argument
  return 0;
}
```

Arrays and pointers

- Most operations on pointers and array looks similar; however, there are some differences
  - array name is not the same as other variables
  - You can store an address in the pointer variable, but you can't store anything in an array name

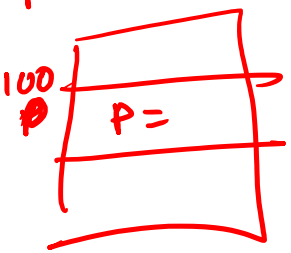
```
int a[10];
int *p = &a[0]; // legal

a = p; // illegal
a = a + 1; // illegal
p = a + 1; // legal
p = p + 1; // legal
```

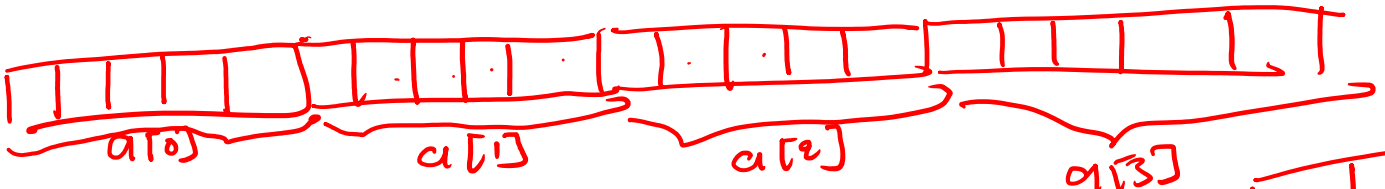
Arrays and pointers

It is a common mistake to assume both two-dimensional array and pointer-to-pointer as the same

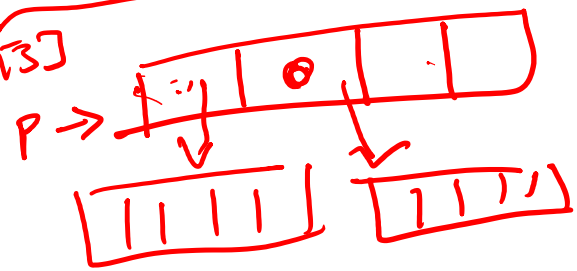
```
int a[4][5];  // a is an array of "5-elements int array"
int **p = (int**)malloc(4 * sizeof(int*));
// p is an array of int*
for (i = 0; i < 4; i++) {
   p[i] = malloc(7 * sizeof(int));
}
```

int **p;





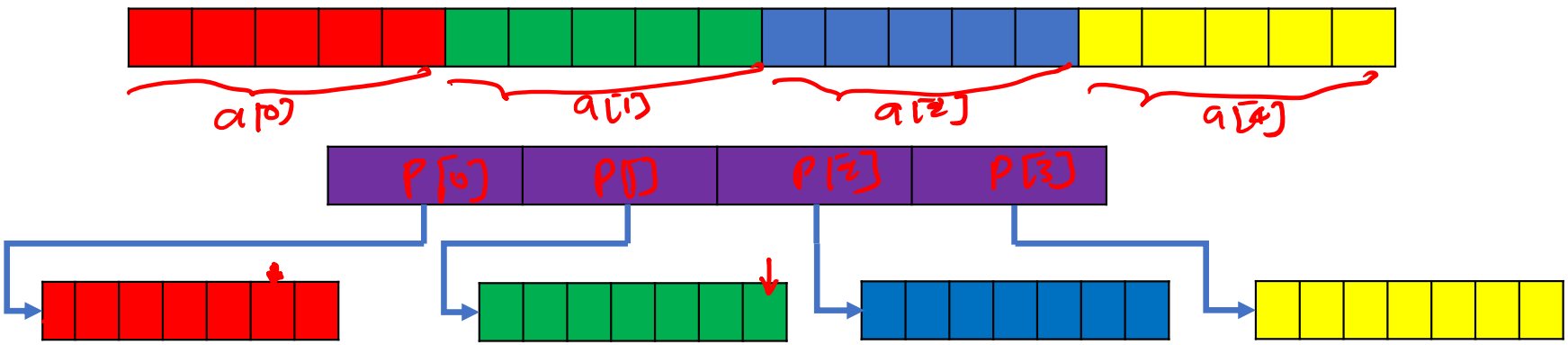
p[1][3]



Arrays and pointers

It is a common mistake to assume both two-dimensional array and pointer-to-pointer as the same

```
int a[4][5];  // a is an array of "5-elements int array"
int **p = (int**)malloc(4 * sizeof(int*));
// p is an array of int*
for (i = 0; i < 4; i++) {
   p[i] = malloc(7 * sizeof(int));
}
```



Arrays and pointers

- Two-dimensional array accesses are different from pointer-to-pointer accesses

```
int a[4][5];  // base address X

int **p = (int**)malloc(4 * sizeof(int*));
int i;
for (i = 0; i < 4; i++) {
   p[i] = malloc(7 * sizeof(int));
}


// &a[2][1] == X + ((5 * 2) + 1) * 4;
// &a[i][j] == X + ((5 * i) + j) * 4;

int *tmp = p[i];
// &p[i][j] == &tmp[j]
```

Does this mean is that pointer to pointer access is expensive than two-dimensional array access?

Homework (ungraded)

- Initialize a pointer-to-pointer (int **p) in a way that the following post-conditions hold

```
int i, j;
for (i = 1; i < 100; i++) {
  assert(p[i] == p[i-1] + 100);
}
for (i = 0; i < 100; i++) {
  assert(i == 0 || p[i][0] > p[i-1][99]);
  for (j = 0; j < 99; j++) {
    assert(p[i][j] < p[i][j+1]);
  }
}
```

Structures

Structure

- Structure is a collection of one or more variables of any type
  - Structure is a user-defined type

- Structure can be used to manage a set of closely related variables efficiently in large programs
  - Structure helps manage closely related variables as a unit rather than as separate entities

Distance between 2D points

```
// returns the distance between two two-dimensional points
double distance2d(int x1, int y1, int x2, int y2) {
   double xpow2 = pow(x2 - x1, 2.0);
   double ypow2 = pow(y2 – y1, 2.0);
   return pow(xpow2 + ypow2, 0.5);
}

int main() {
   int x1 = 2, y1 = 2;  // point-1 coordinates
   int x2 = 3, y2 = 5;  // point-2 coordinates
   double r = distance2d(x1, y1, x2, y2);
   printf(“%lf\n”, r);
   return 0;
}
```

Distance between 3D points

```
// returns the distance between two three-dimensional points
double distance3d(int x1, int y1, int z1, int x2, int y2, int z2) {
   double xpow2 = pow(x2 - x1, 2.0);
   double ypow2 = pow(y2 – y1, 2.0);
   double zpow2 = pow(z2 – z1, 2.0);
   return pow(xpow2 + ypow2 + zpow2, 0.5);
}

int main() {
   int x1 = 2, y1 = 2, z1 = 2; // point-1 coordinates
   int x2 = 3, y2 = 5, z2 = 8; // point-2 coordinates
   double r = distance3d(x1, y1, z1, x2, y2, z2);
   printf(“%lf\n”, r);
   return 0;
}
```

Structures

- A large program would be challenging to manage if we just use the primitive types

Distance between 3D points

```
// returns the distance between two three-dimensional points
double distance3d(int x1, int y1, int z1, int x2, int y2, int z2) {
   double xpow2 = pow(x2 - x1, 2.0);
   double ypow2 = pow(y2 – y1, 2.0);
   double zpow2 = pow(z2 – z1, 2.0);
   return pow(xpow2 + ypow2 + zpow2, 0.5);
}

int main() {
   int x1 = 2, y1 = 2, z1 = 2; // point-1 coordinates
   int x2 = 3, y2 = 5, z2 = 8; // point-2 coordinates
   double r = distance3d(x1, y1, z1, x2, y2, z2);
   printf(“%lf\n”, r);
   return 0;
}
```

3D points using structure

```
// returns the distance between two 3D points
double distance3d(struct point3d pt1,
                  struct point3d pt2)
{
   double xpow2 = pow(pt2.x – pt1.x, 2.0);
   double ypow2 = pow(pt2.y – pt1.y, 2.0);
   double zpow2 = pow(pt2.z – pt1.z, 2.0);
   return pow(xpow2 + ypow2 + zpow2, 0.5);
}

int main() {
   int x1 = 2, y1 = 2, z1 = 2; // point-1
   int x2 = 3, y2 = 5, z2 = 8; // point-2
   struct point3d p1 = make_point3d(x1, y1, z1);
   struct point3d p2 = make_point3d(x2, y2, z2);

   double r = distance3d(p1, p2);
   printf("%lf\n", r);
   return 0;
}
```

```
struct point3d {
    int x;
    int y;
    int z;
};

struct point3d make_point3d(int x,
                            int y,
                            int z)
{
   struct point3d t;
   t.x = x;
   t.y = y;
   t.z = z;
   return t;
}
```

3D points using structures

struct point3d p = make_point3d(0, 1, 2);

```
struct point3d {
    int x;
    int y;
    int z;
};

struct point3d make_point3d(int x,
                            int y,
                            int z)
{
   struct point3d t;
   t.x = x;
   t.y = y;
   t.z = z;
   return t;
}
```

Parameter passing

```
// returns the distance between two 3D points
double distance3d(struct point3d pt1,
                  struct point3d pt2)
{
   double xpow2 = pow(pt2.x – pt1.x, 2.0);
   double ypow2 = pow(pt2.y – pt1.y, 2.0);
   double zpow2 = pow(pt2.z – pt1.z, 2.0);
   return pow(xpow2 + ypow2 + zpow2, 0.5);
}

int main() {
   int x1 = 2, y1 = 2, z1 = 2; // point-1
   int x2 = 3, y2 = 5, z2 = 8; // point-2
   struct point3d p1 = make_point3d(x1, y1, z1);
   struct point3d p2 = make_point3d(x2, y2, z2);

   double r = distance3d(p1, p2);
   printf("%lf\n", r);
   return 0;
}
```

Parameter passing

We can pass a pointer to the structure to avoid copy.

```
// returns the distance between two 3D points
double distance3d(struct point3d *pt1,
                  struct point3d *pt2)
{
   double xpow2 = pow(pt2->x – pt1->x, 2.0);
   double ypow2 = pow(pt2->y – pt1->y, 2.0);
   double zpow2 = pow(pt2->z – pt1->z, 2.0);
   return pow(xpow2 + ypow2 + zpow2, 0.5);
}

int main() {
   int x1 = 2, y1 = 2, z1 = 2; // point-1
   int x2 = 3, y2 = 5, z2 = 8; // point-2
   struct point3d p1 = make_point3d(x1, y1, z1);
   struct point3d p2 = make_point3d(x2, y2, z2);

   double r = distance3d(&p1, &p2);
   printf("%lf\n", r);
   return 0;
}
```

Linked lists

References

- Read section-10.2 from the Cormen et al. book

Issues with arrays

- Inserting a value into an array may lake a large number of operations
- Deleting an element from an array may a take large number of operations
- Extending an array may not always be possible

Inserting an element

3	9	13	17	18	23	29	33	39	47	55	61	73	84	88	92	94	96	98	99

Insert 40 in a way that the target array remains sorted.

Inserting an element

3	9	13	17	18	23	29	33	39	47	55	61	73	84	88	92	94	96	98	99

Insert 40 in a way that the target array remains sorted.

3	9	13	17	18	23	29	33	39	40	47	55	61	73	84	88	92	94	96	98	99

Deleting an element

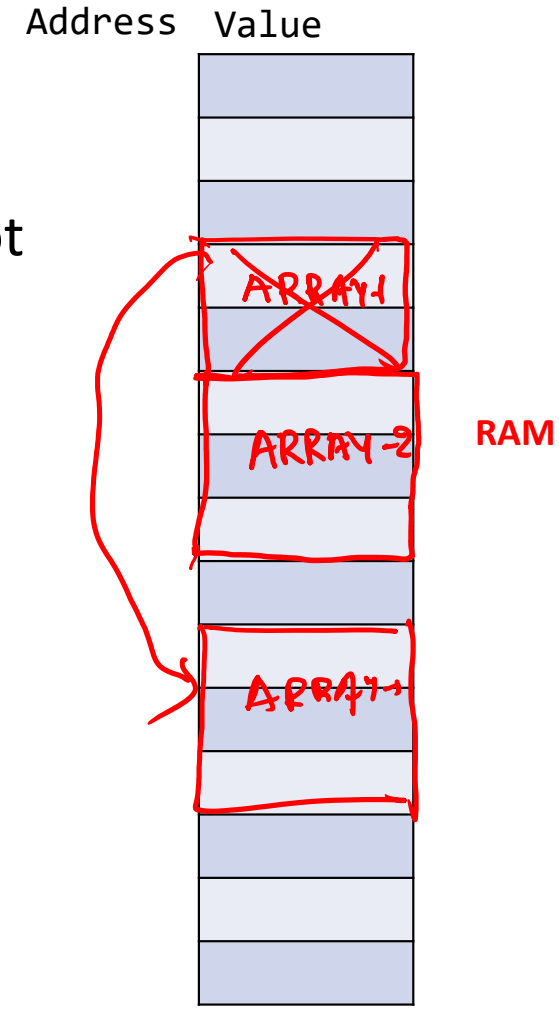
3	9	13	17	18	23	29	33	39	~~47~~	55	61	73	84	88	92	94	96	98	99

Delete 47 in a way that the target array remains sorted.

Extending an array

- Consecutive address locations may not be available to extend an array

a = malloc( )
p = a;
p1 = a;
p3 = p1;



Eliminating array copy

- Why do we need to copy during array insertion/deletion?
  - What is the fundamental problem?

- Why can’t we copy an array to a different location when consecutive addresses are not available during extension?

Eliminating array copy

- Why do we need to copy during array insertion/deletion?
  - What is the fundamental problem?
    - The array elements must be stored at consecutive addresses
- Why can't we copy an array to a different location when consecutive addresses are not available during extension?
  - The address of the array may be stored in other pointer variables
  - We need to update all of them if we want to relocate an array

Eliminating array copy

- Linked list is almost like an array, except the addresses of two consecutive elements in the list may not be consecutive
- Thus, a linked list doesn’t require copying of elements during insertion and deletion
- Linked list always supports extension if the memory is available
- The elements of a linked list are also called nodes

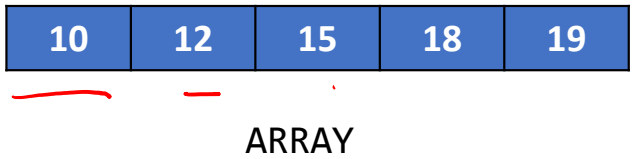
Linked list

- Suppose the base address of an array is stored in the variable `ptr`
- How can we access the fifth element of the array?

- Suppose the base address of a linked list is stored in the variable `ptr`
- How can we access the fifth element of the linked list?

Linked list

- Every element in a linked list also stores the address of the next node



ARRAY

Type of a linked list node

```
struct node {
    int val;
    struct node *next;
};
```

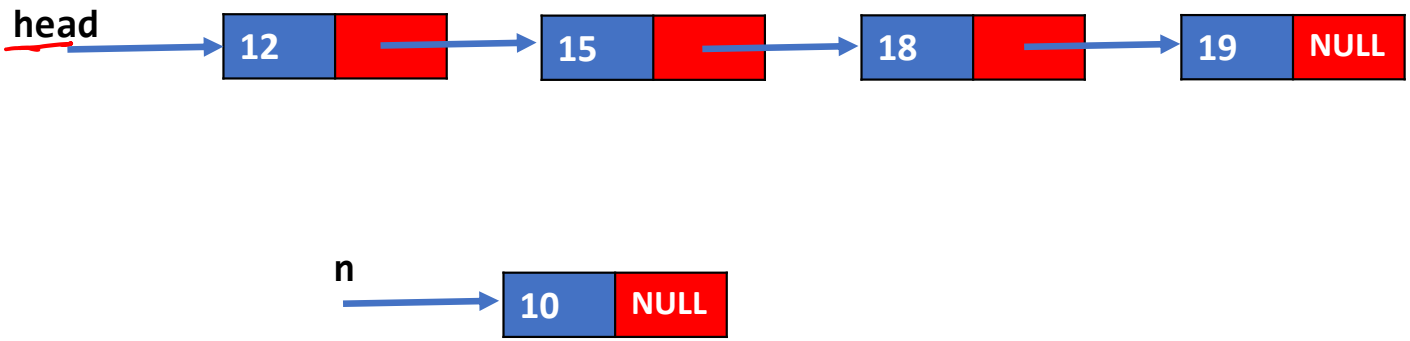
Allocation

```
struct node {
  int val;
  struct node *next;
};

struct node* allocate_node(int val) {
  struct node *n = (struct node*)malloc(sizeof(struct node));
  // handle it gracefully instead of assertion failures
  assert(n != NULL);
  n->val = val;
  n->next = NULL;
  return n;
}
```

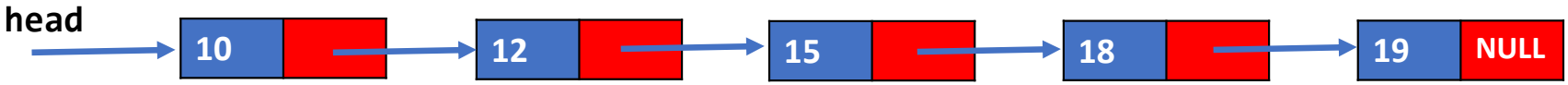
Allocating a linked list node.

Insertion (front)

- Before insertion



- After insertion



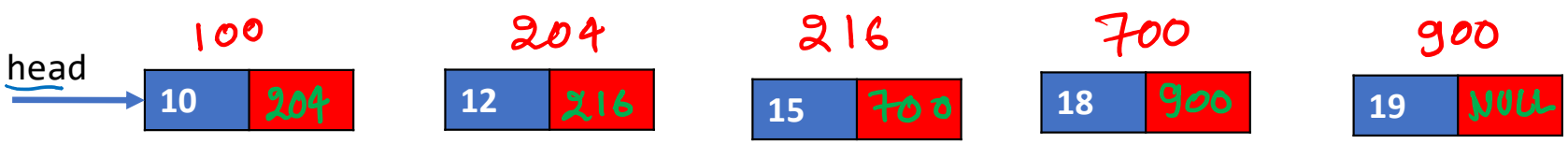
Today’s topics

- Linked lists

Linked lists

Linked list

- Every element in a linked list also stores the address of the next node
- head points to the first element of the linked list



Type of a linked list node

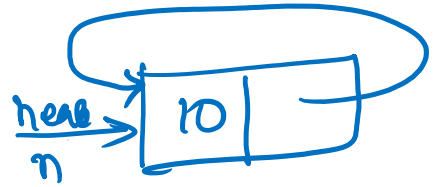
```
struct node {
    int val;
    struct node *next;
};

struct node *head;
```

Allocation

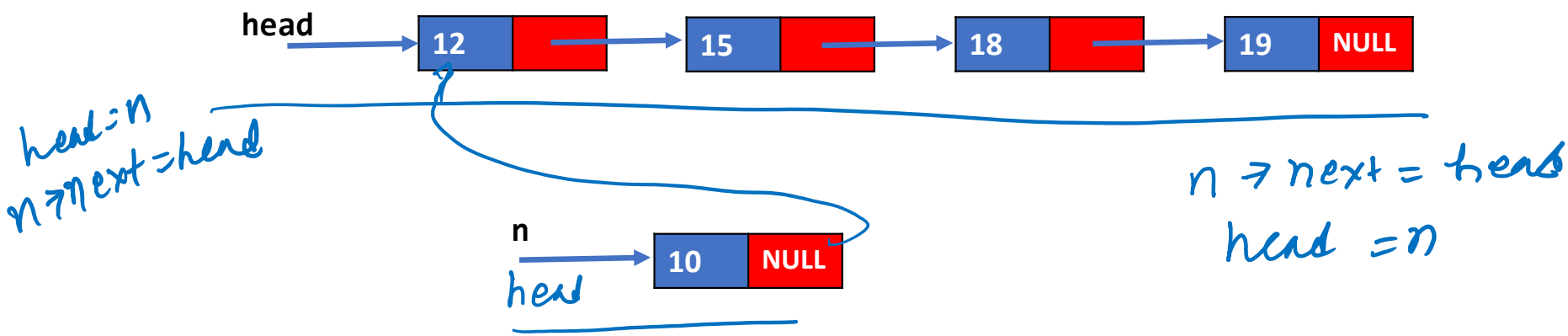
```
struct node {
  int val;
  struct node *next;
};

struct node* allocate_node(int val) {
  struct node *n = (struct node*)malloc(sizeof(struct node));
  // handle it gracefully instead of assertion failures
  assert(n != NULL);
  n->val = val;
  n->next = NULL;
  return n;
}
```

Allocating a linked list node.

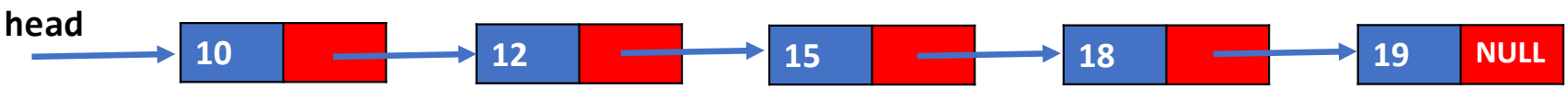
Insertion (front)



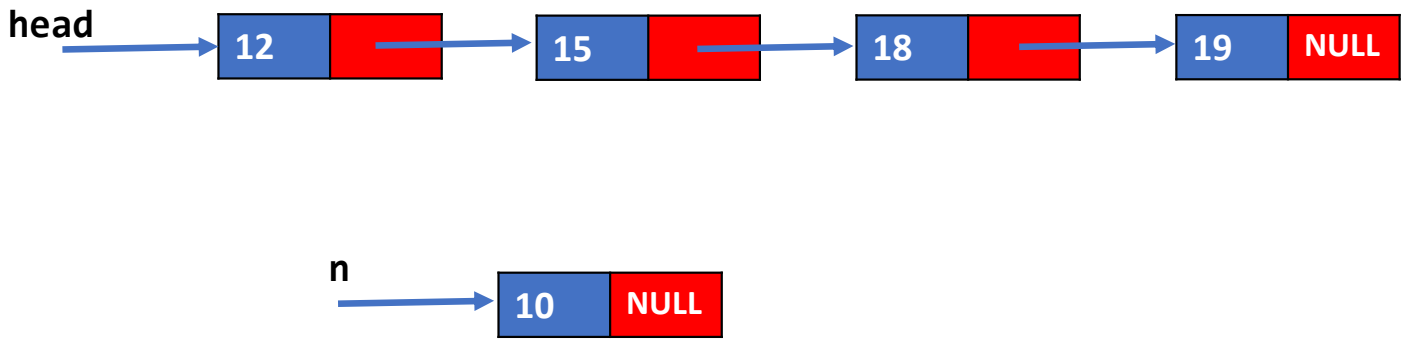
- Before insertion



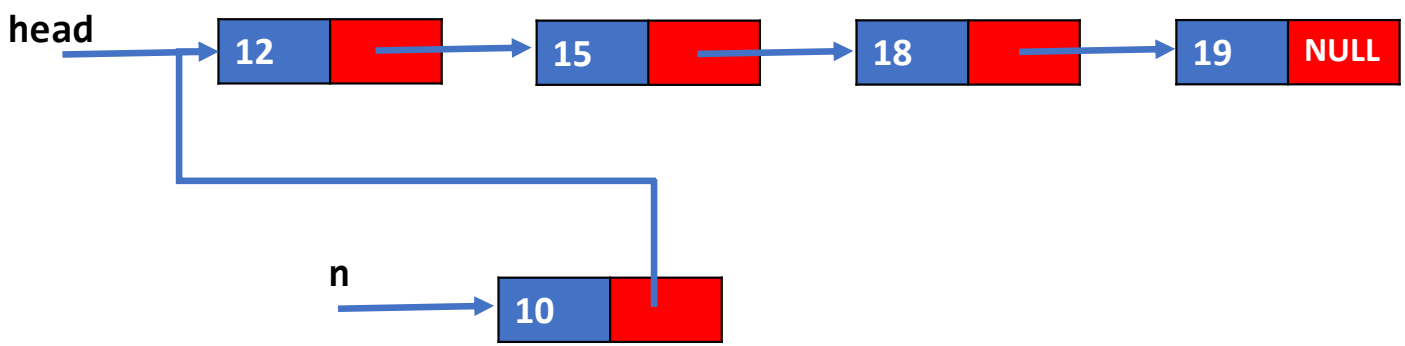
- After insertion



Insertion (front)

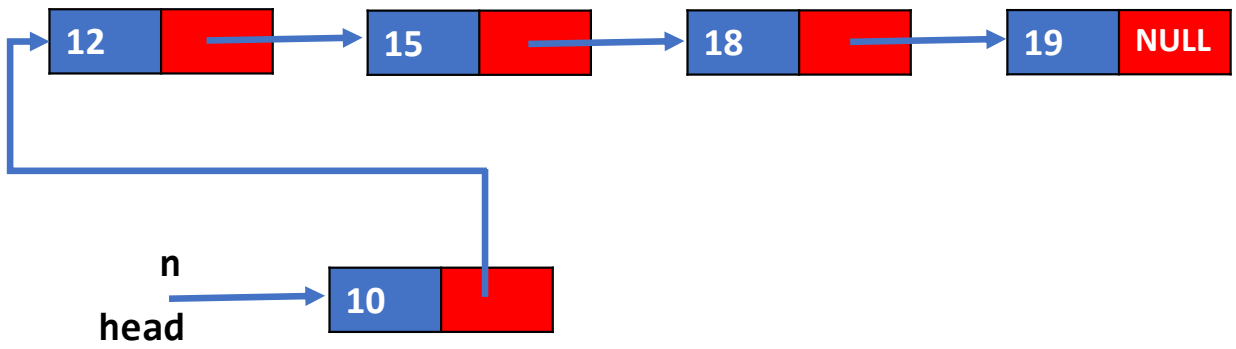


Insertion (front)



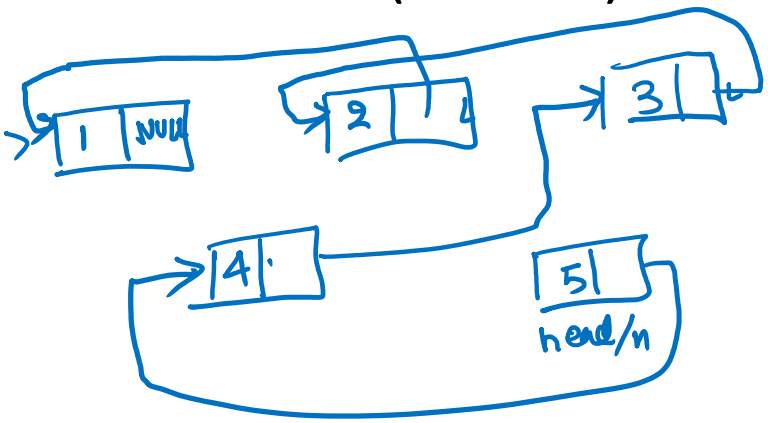

```
n->next = head;
```

Insertion (front)




```
n->next = head;
head = n
```

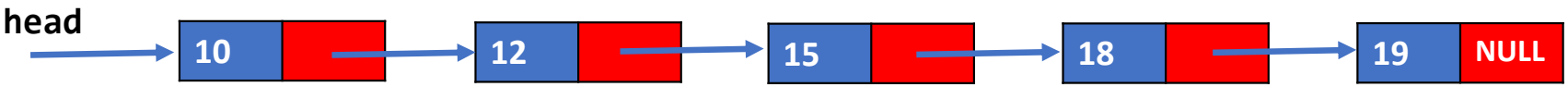
Insertion (front)



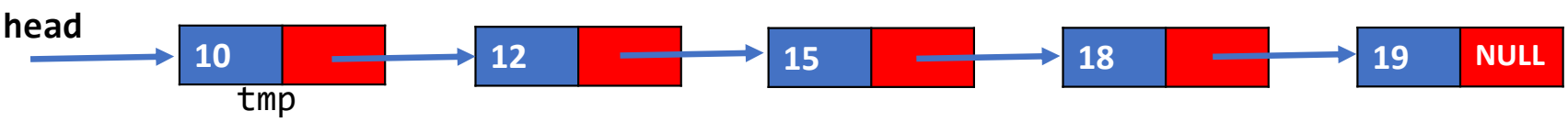

```
struct node {
  int val;
  struct node *next;
};

struct node* insert_front(struct node *head,
                          struct node *n) {
  n->next = head;
  return n;  // returns the new head
}

int main() {
// head points to the first element of list
  struct node *head = allocate_node(1);
  struct node *n;
  int i;

  for (i = 2; i <= 5; i++) {
    n = allocate_node(i);
    head = insert_front(head, n);
  }
  print_list(head);
  return 0;
}
```

Time complexity of insert_front: O(1)

Printing list

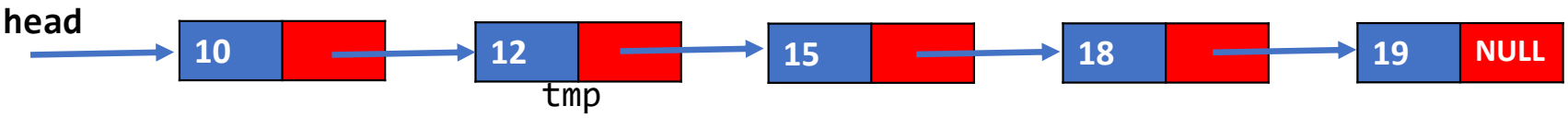


Printing list



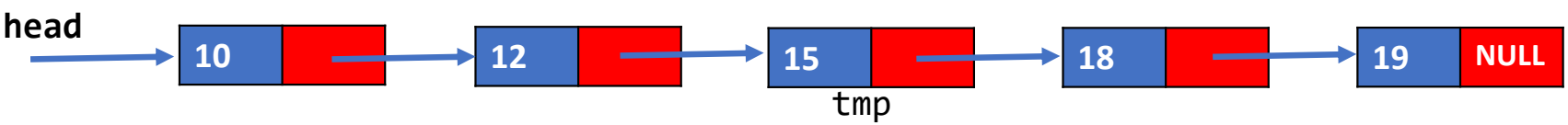

```
tmp = head
printf("%d ", tmp->val); 10
```

Printing list



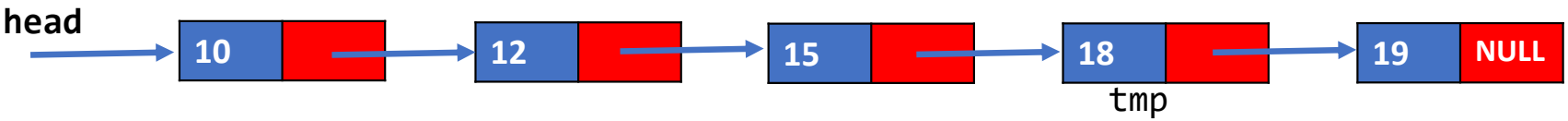

```
tmp = head
printf("%d ", tmp->val); 10
tmp = tmp->next;
printf("%d ", tmp->val); 12
```

Printing list



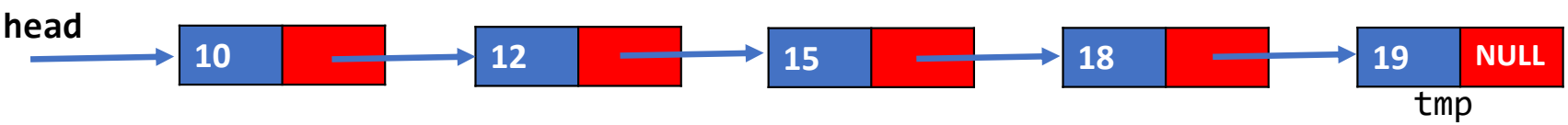

```
tmp = head
printf("%d ", tmp->val); 10
tmp = tmp->next;
printf("%d ", tmp->val); 12
tmp = tmp->next;
printf("%d ", tmp->val); 15
```

Printing list




```
tmp = head
printf("%d ", tmp->val); 10
tmp = tmp->next;
printf("%d ", tmp->val); 12
tmp = tmp->next;
printf("%d ", tmp->val); 15
tmp = tmp->next;
printf("%d ", tmp->val); 18
```

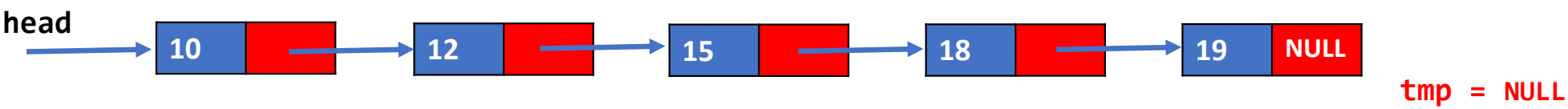
Printing list




```
tmp = head
printf("%d ", tmp->val); 10
tmp = tmp->next;
printf("%d ", tmp->val); 12
tmp = tmp->next;
printf("%d ", tmp->val); 15
tmp = tmp->next;
printf("%d ", tmp->val); 18
```

```
tmp = tmp->next;
printf("%d ", tmp->val); 19
```

Printing list

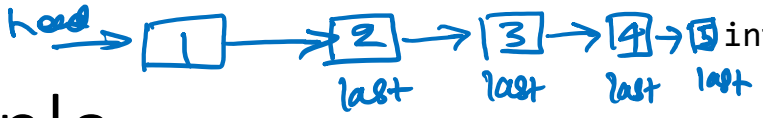



```
tmp = head
printf("%d ", tmp->val); 10
tmp = tmp->next;
printf("%d ", tmp->val); 12
tmp = tmp->next;
printf("%d ", tmp->val); 15
tmp = tmp->next;
printf("%d ", tmp->val); 18
```

```
tmp = tmp->next;
printf("%d ", tmp->val); 19
tmp = tmp->next
```

Printing list

```
struct node {
  int val;
  struct node *next;
};

void print_list(struct node *head) {
  struct node *tmp = head;
  while (tmp != NULL) {
     printf("%d ", tmp->val);
     tmp = tmp->next;
  }
  printf("\n");
}
```



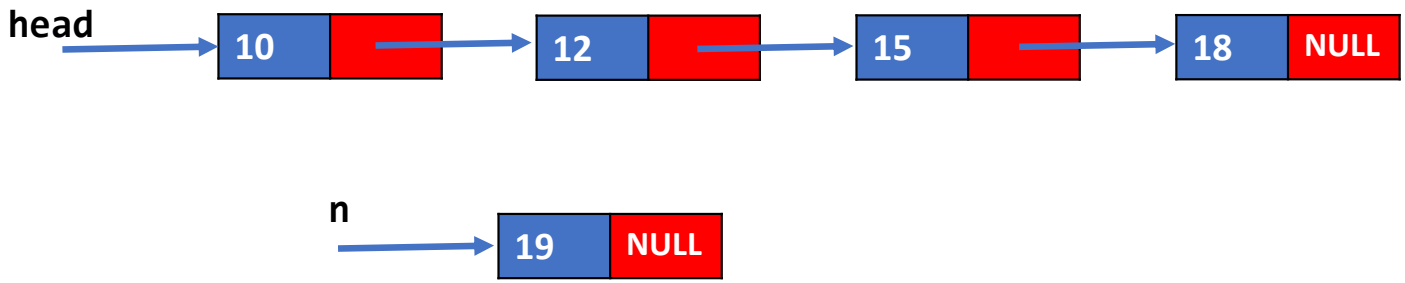
Example

```c
struct node {
  int val;
  struct node *next;
};

struct node* allocate_node(int val)
{
  struct node *n =
     (struct node*)malloc(sizeof(struct node));
  assert(n != NULL);
  n->val = val;
  n->next = NULL;
  return n;
}
```

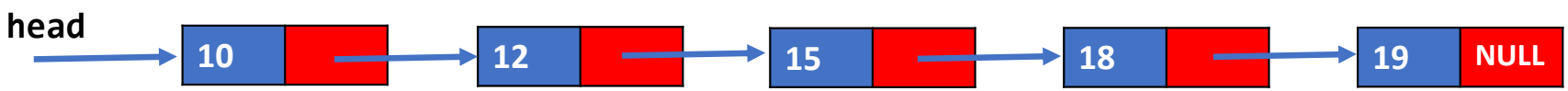
```c
int main() {
  struct node *head = allocate_node(1);
  struct node *last = head;
  int i, sum = 0;

  for (i = 2; i <= 5; i++) {
    struct node *n = allocate_node(i);
    last->next = n;
    last = n;
  }

  /* printing values of all nodes in the list */
  while (head != NULL) {
    printf("%d ", head->val);
    head = head->next;
  }

  while (head != NULL) {
    sum += head->val;
    head = head->next;
  }
  printf("sum of all nodes is : %d\n", sum);
  /* what would be the value of sum? */
  return 0;
}
```

Insertion (rear)

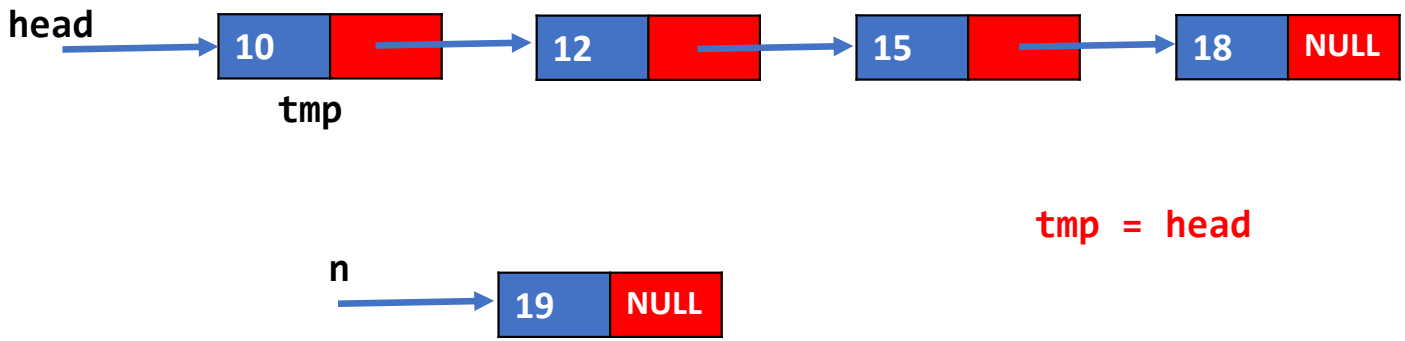
- Before insertion



- After insertion

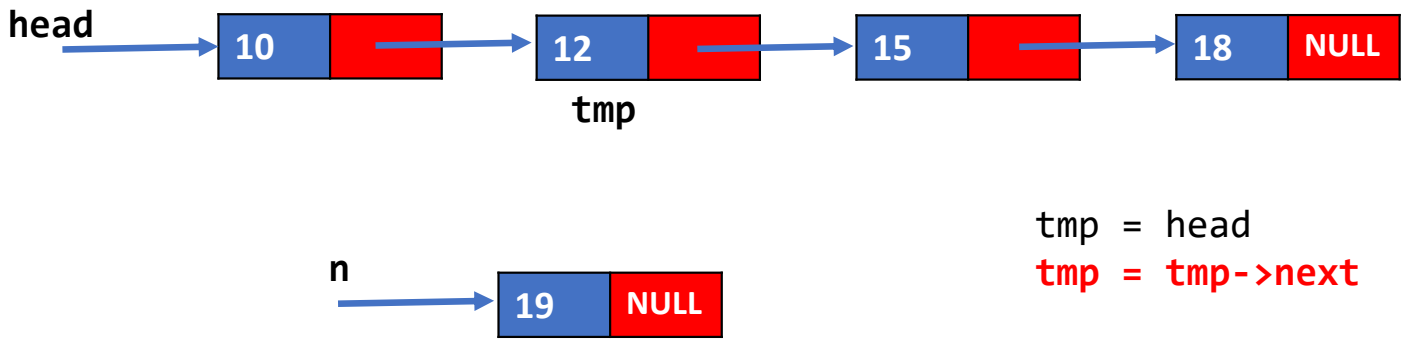


Insertion (rear)



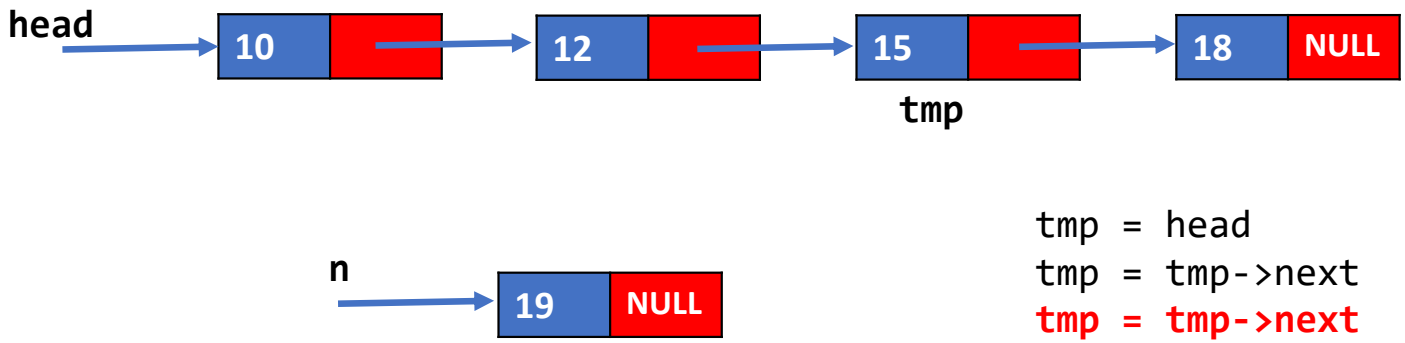

```
tmp = head
```

Insertion (rear)




```
tmp = head
tmp = tmp->next
```

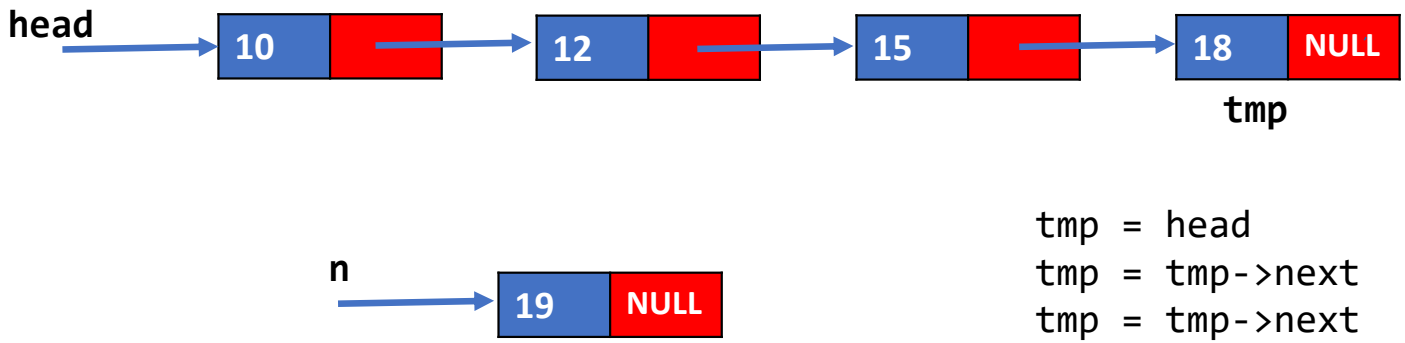
Insertion (rear)




```
tmp = head
tmp = tmp->next
tmp = tmp->next
```

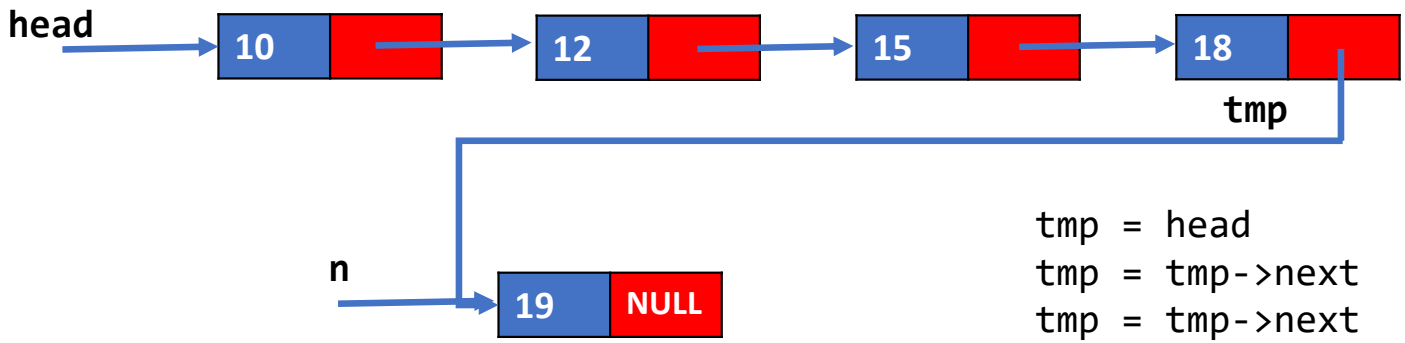
Insertion (rear)

tmp → next = n



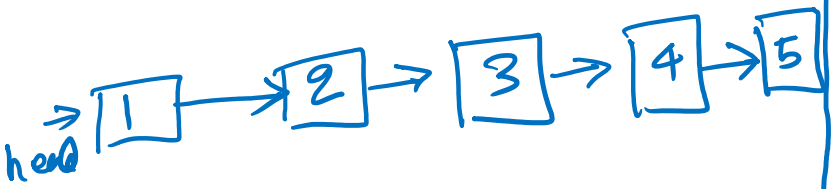

```
tmp = head
tmp = tmp->next
tmp = tmp->next
tmp = tmp->next
```

Insertion (rear)




```
tmp = head
tmp = tmp->next
tmp = tmp->next
tmp = tmp->next
tmp->next = n
```

Insertion (rear)

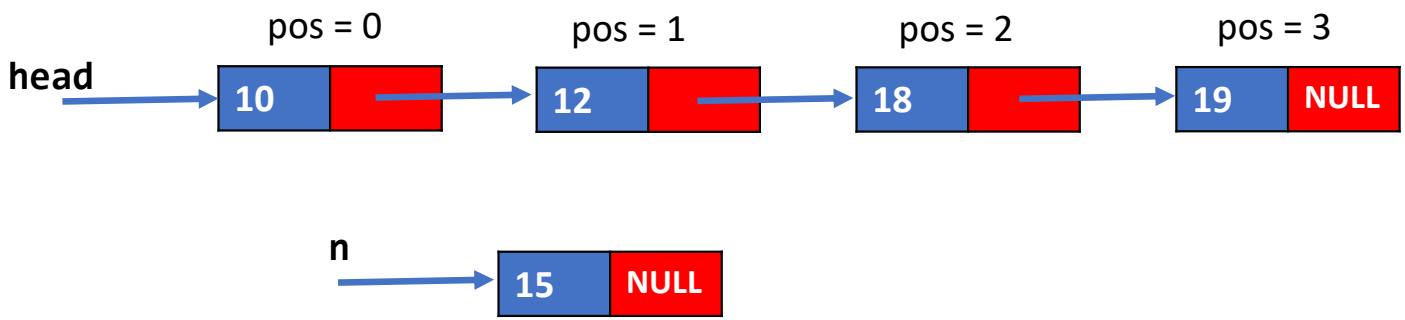



```
void insert_rear(struct node *head, struct node *n) {
  assert(head != NULL);
  struct node *tmp = head;
  while (tmp->next != NULL) {
     tmp = tmp->next;
  }
  tmp->next = n;
}

int main() {
// head points to the first element of list
  struct node *head = allocate_node(1);
  struct node *n;
  int i;

  for (i = 2; i <= 5; i++) {
    n = allocate_node(i);
    insert_rear(head, n);
  }
  print_list(head);
  return 0;
}
```

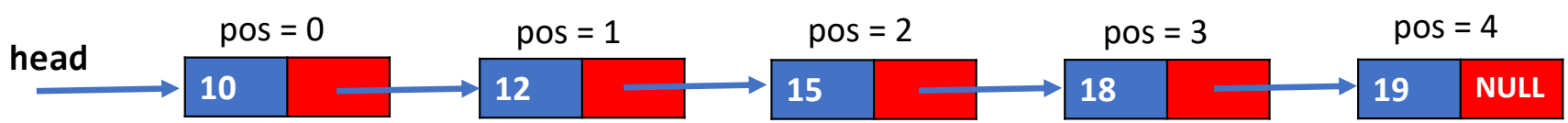
Time complexity of insert_back: O(n)

Insertion (at specific position)

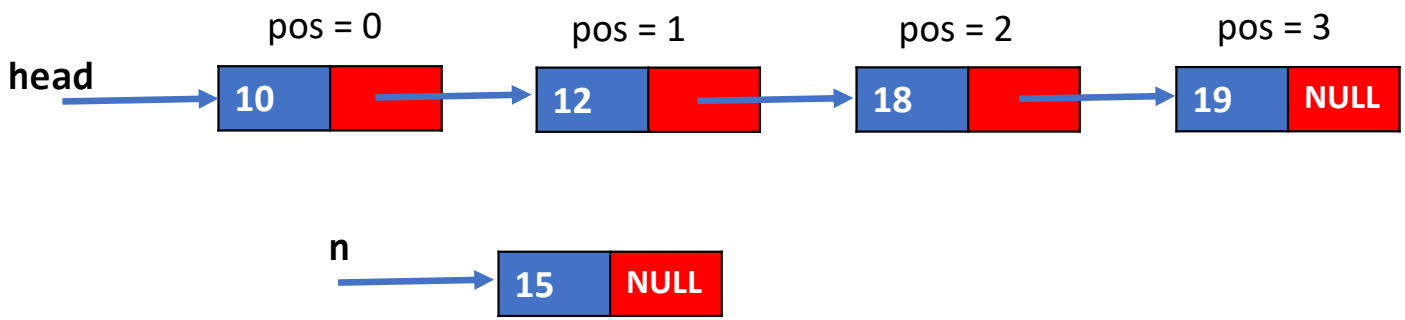
- Before insertion



- After inserting "n" at position two

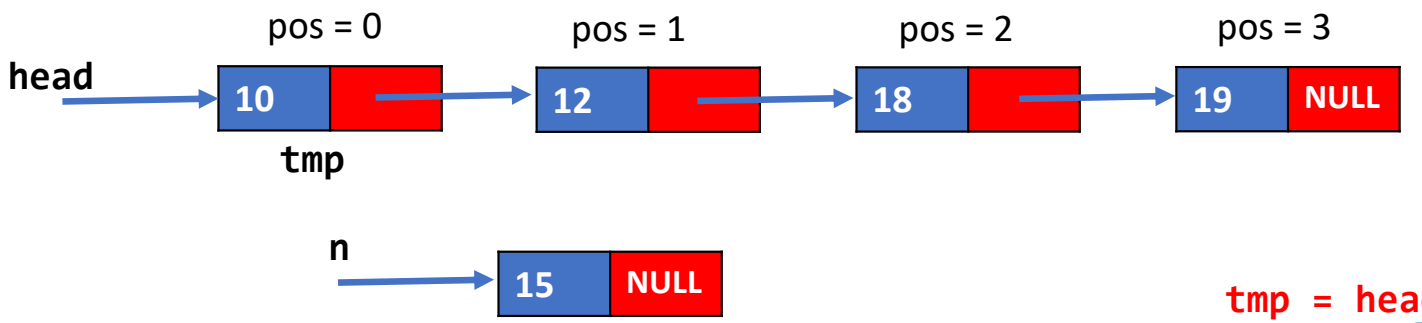


Insertion (at specific position)



Insert the node n at position two in the linked list.

Insertion (at specific position)



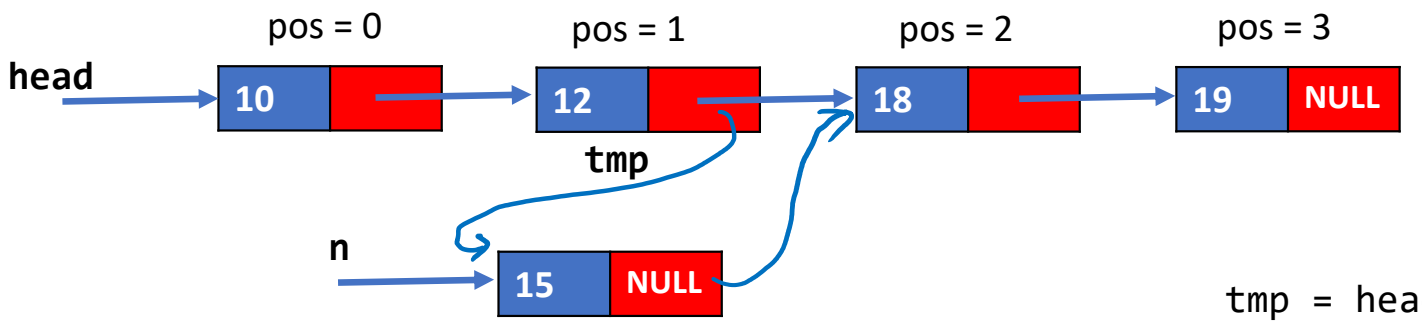

```
tmp = head
pos = 0
```

Insertion (at specific position)

tmp1 = tmp->next
tmp->next = n
n->next = tmp1

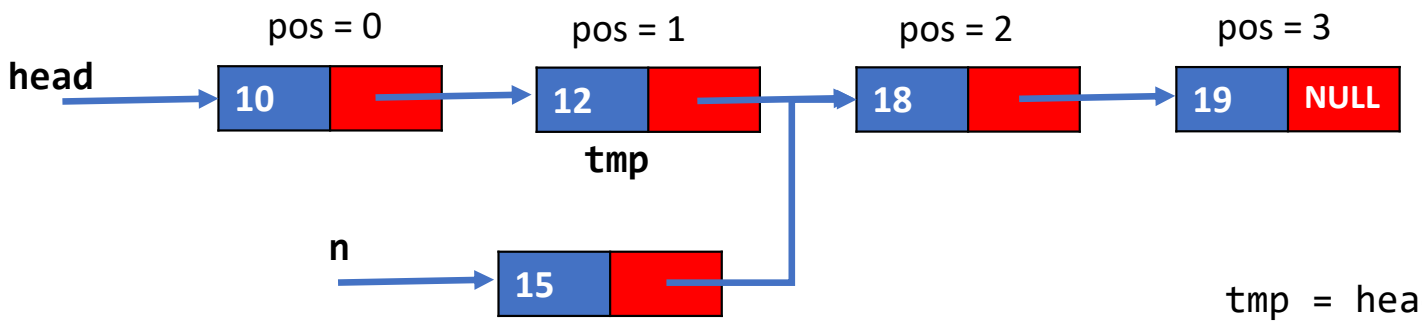
~~tmp->next = n~~
~~n->next = tmp->next~~

n->next = tmp->next
tmp->next = n



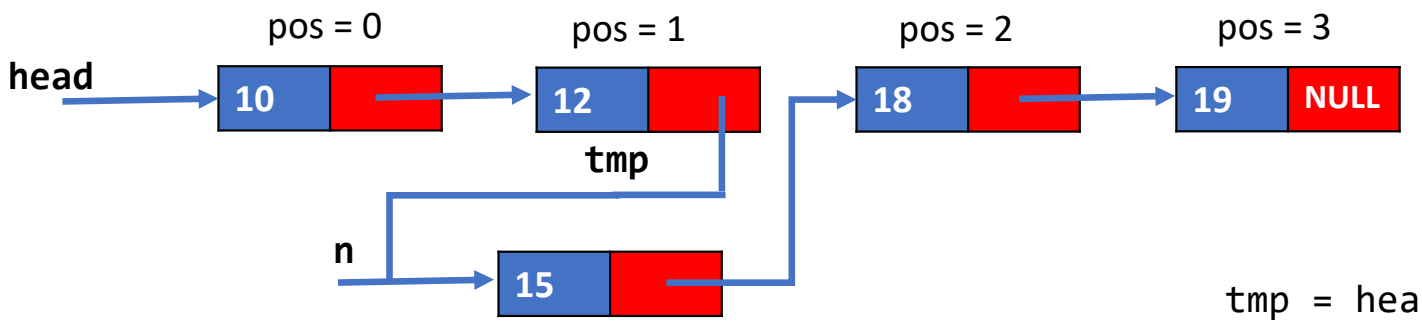

```
tmp = head
pos = 0
tmp = tmp->next
pos = 1
```

Insertion (at specific position)



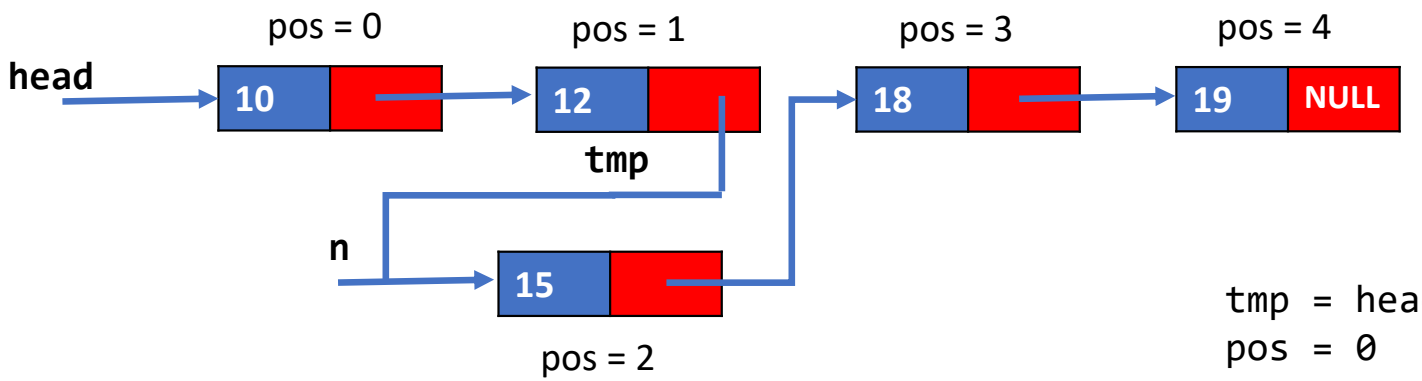

```
tmp = head
pos = 0
tmp = tmp->next
pos = 1
n->next = tmp->next
```

Insertion (at specific position)




```
tmp = head
pos = 0
tmp = tmp->next
pos = 1
n->next = tmp->next
tmp->next = n
```

Insertion (at specific position)

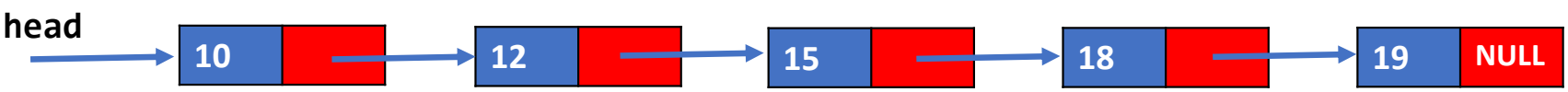



```
tmp = head
pos = 0
tmp = tmp->next
pos = 1
n->next = tmp->next
tmp->next = n
```

Insertion (at specific position)

```
void insert_pos(struct node *head, struct node *n, int pos) {
  struct node *tmp = head;
  int i = 0;
  assert(pos > 0);
  while (i != pos – 1 && tmp != NULL) {
    tmp = tmp->next;
    i = i + 1;
  }
  if (tmp != NULL) {
    n->next = tmp->next;
    tmp->next = n;
  }
}
```

Search



Search a linked list node that contains 18 in the value field.
If the node exists, the search routine returns the node; otherwise, it returns NULL. The search operation doesn’t modify the linked list.

Time complexity:

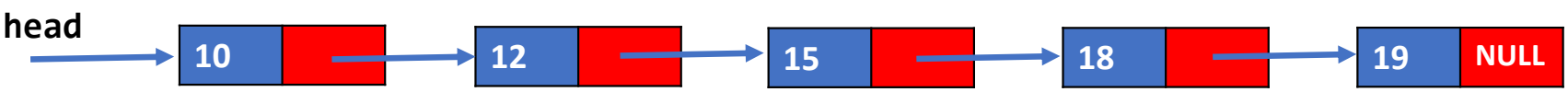
Search

```
// Returns a linked list (starts at head) node that contains
// the input argument val

struct node* search(struct node *head, int val) {
  struct node *tmp = head;
  while (tmp != NULL) {
    if (tmp->val == val) {
      return tmp;
    }
    tmp = tmp->next;
  }
  return NULL;
}
```

O(N)

Search using recursion

- `struct node* search(struct node *head, int val);`



Base case?
Recursive step?

Search a linked list node that contains 18 in the value field.
If the node exists, the search routine returns the node; otherwise, it returns NULL. The search operation doesn't modify the linked list.

Time complexity:

Search using recursion

- `struct node* search(struct node *head, int val);`
- Base case:
  - If the list is empty (`NULL`), return `NULL`
  - If the value in the first node (`head`) matches with the value being searched (`val`), return `head`
- Recursive case:
  - Recursively search `val` in the linked list starting from the next node

Search using recursion

```
// Returns a linked list (starts at head) node that contains
// the input argument val

struct node* search_rec(struct node *head, int val) {
  if (head == NULL) {
    return NULL;
  }
  if (head->val == val) {
    return head;
  }
  return search_rec(head->next, val);
}
```

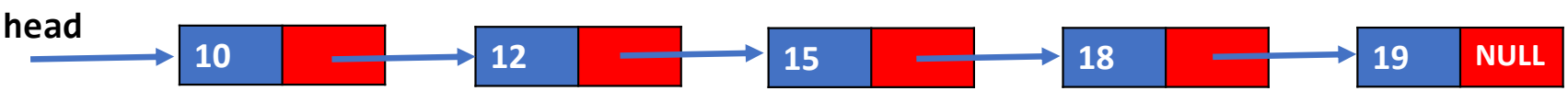
$T(n) = T(n-1) + c$

$O(n)$

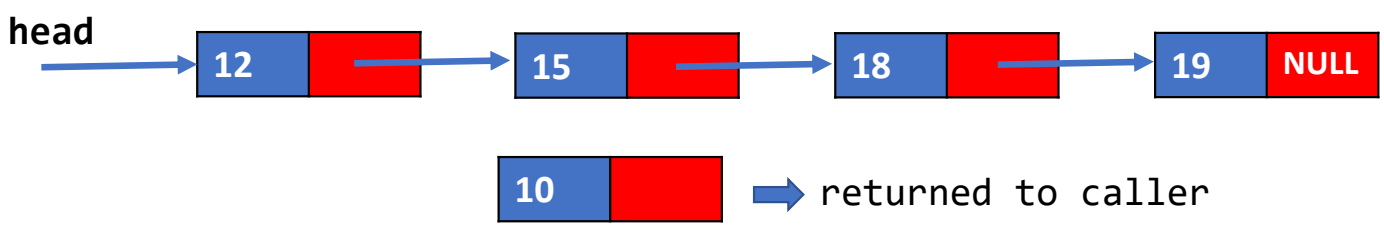
Delete (front)

tmp = head;
head = head->next
return tmp

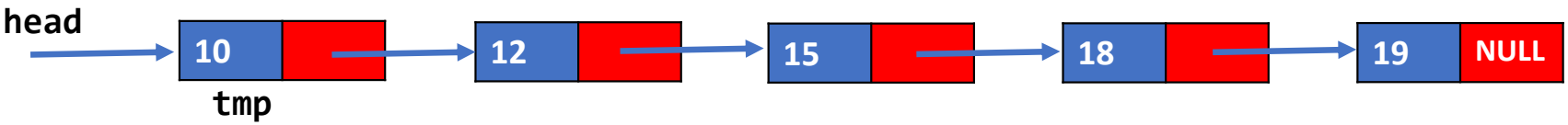
- Before delete



- After delete

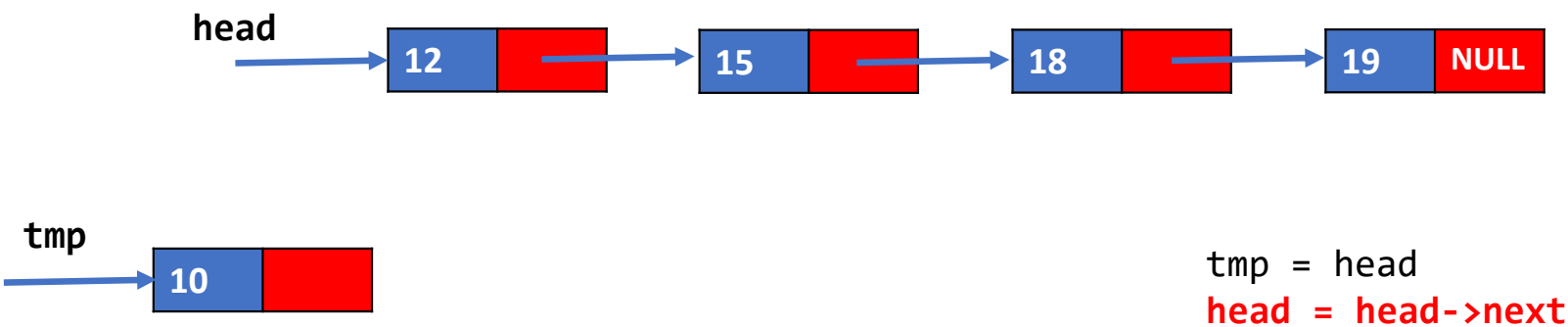


Delete (front)



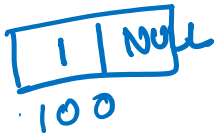

```
tmp = head
```

Delete (front)

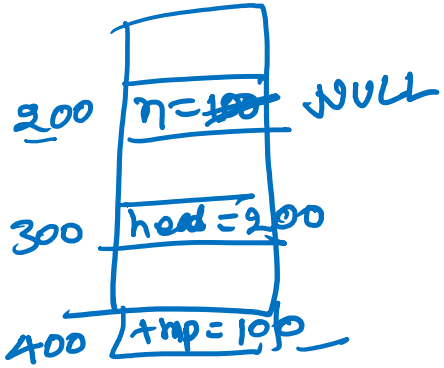



```
tmp = head
head = head->next
```

Delete (front)

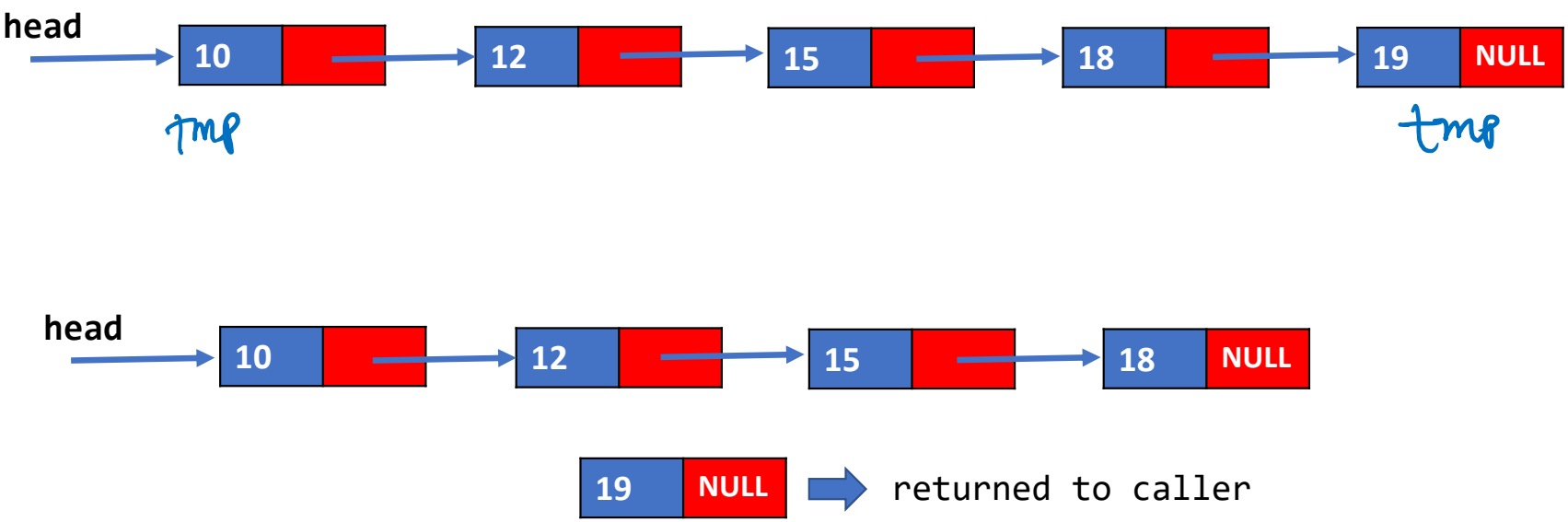



```
main()
{
    struct node *n = allocate_node();
    delete_front(&n);
    printf("%p", n);
}
```

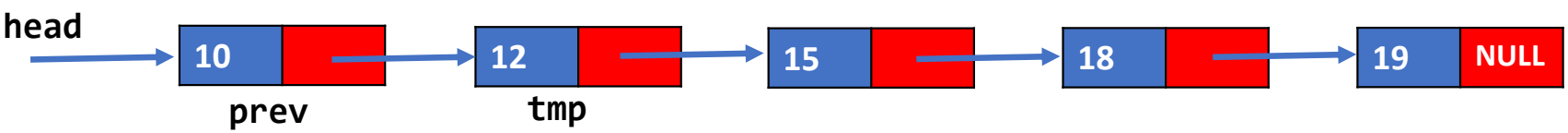



```
// update the head
// return the deleted node

struct node *delete_front(struct node **head) {
  struct node *tmp = *head;
  assert(tmp != NULL);
  *head = tmp->next;
  return tmp;
}
```

Delete (rear)

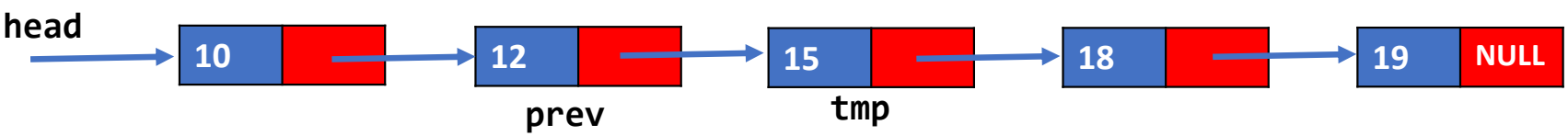

head
10
12
15
18
19
NULL
tmp
tmp
head
10
12
15
18
NULL
19
NULL
returned to caller

Delete (rear)



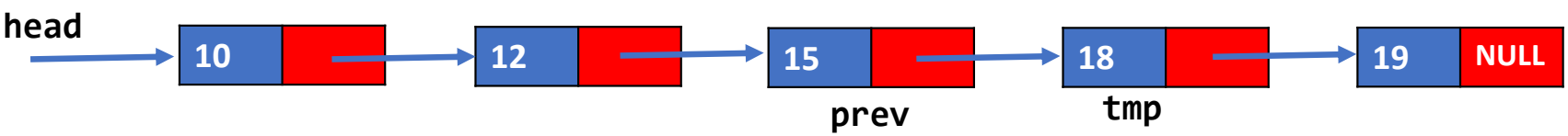

```
prev = head
tmp = head->next
```

Delete (rear)




```
prev = head
tmp = head->next
prev = tmp
tmp = tmp->next
```

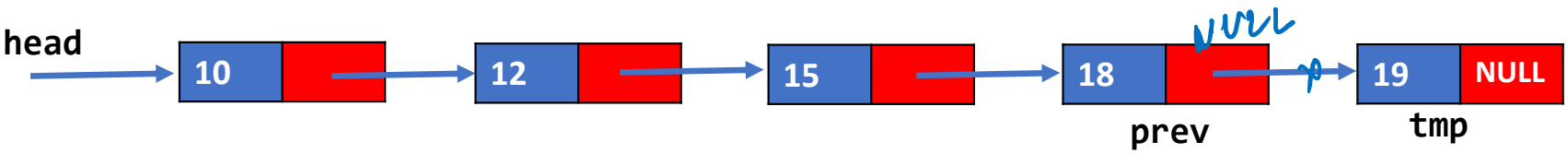
Delete (rear)




```
prev = head
tmp = head->next
prev = tmp
tmp = tmp->next
prev = tmp
tmp = tmp->next
```

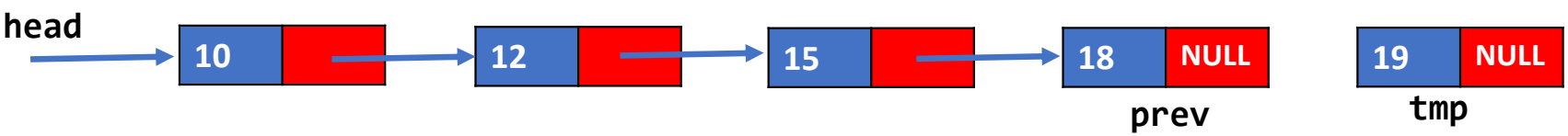
Delete (rear)

prev->next = NULL
tmp




```
prev = head
tmp = head->next
prev = tmp
tmp = tmp->next
prev = tmp
tmp = tmp->next
prev = tmp
tmp = tmp->next
```

Delete (rear)




```
prev = head
tmp = head->next
prev = tmp
tmp = tmp->next
prev = tmp
tmp = tmp->next
prev = tmp
tmp = tmp->next
prev->next = NULL
```

Delete (rear)

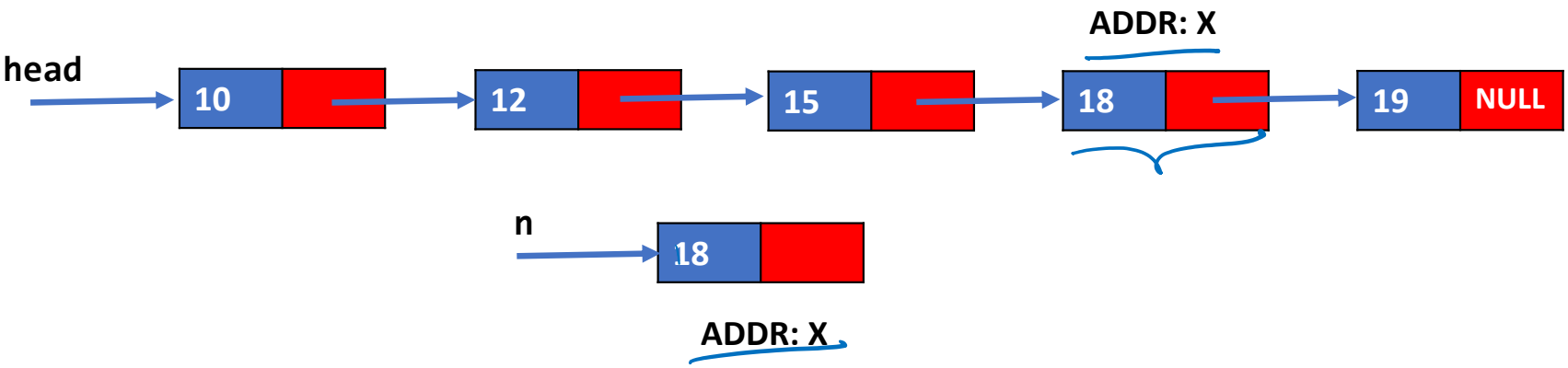
```
struct node *delete_rear(struct node **head) {
  struct node *tmp = *head;
  assert(tmp);
  if (tmp->next == NULL) {
    *head = NULL;  // single-node list
    return tmp;
  }
  struct node *prev = tmp;
  tmp = tmp->next;
  while (tmp->next != NULL) {
    prev = tmp;
    tmp = tmp->next;
  }
  prev->next = NULL;
  return tmp;
}
```

Search returns a linked list node

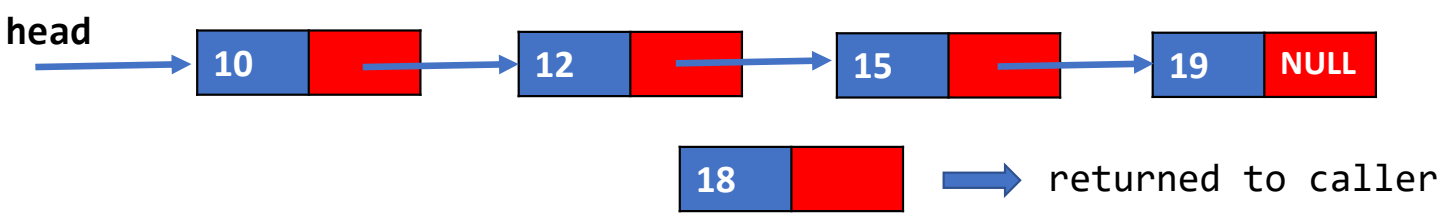
```
// Returns a linked list (starts at head) node that contains
// the input argument val

struct node* search(struct node *head, int val) {
  struct node *tmp = head;
  while (tmp != NULL) {
    if (tmp->val == val) {
      return tmp;
    }
    tmp = tmp->next;
  }
  return NULL;
}
```

Delete a given node

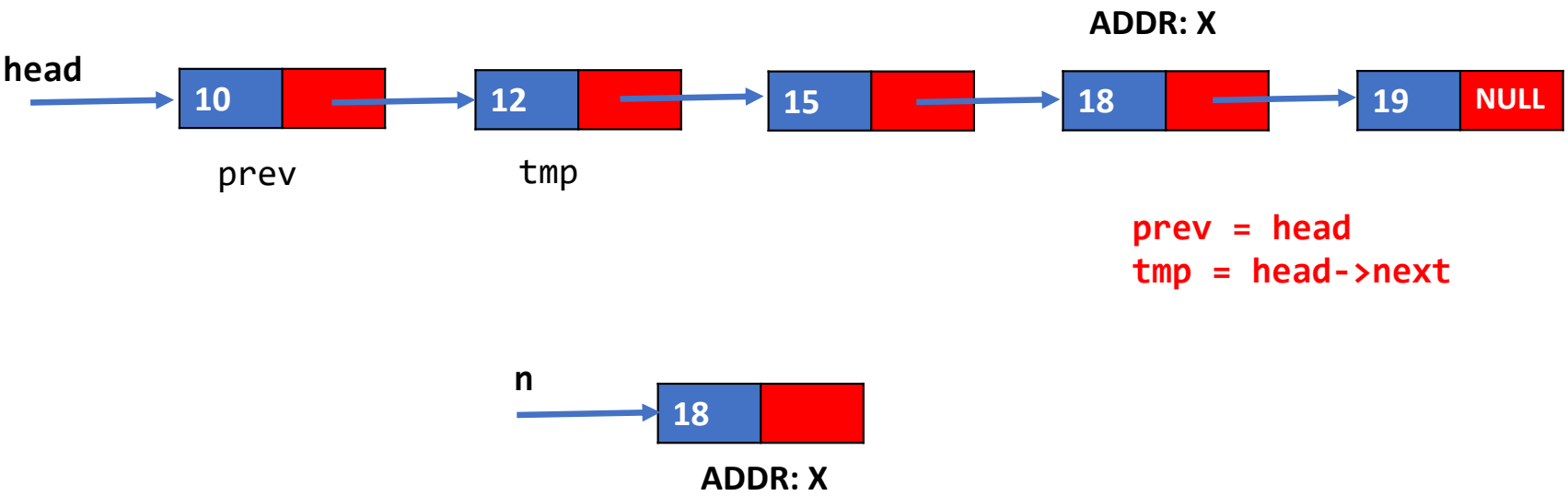
- Before deletion



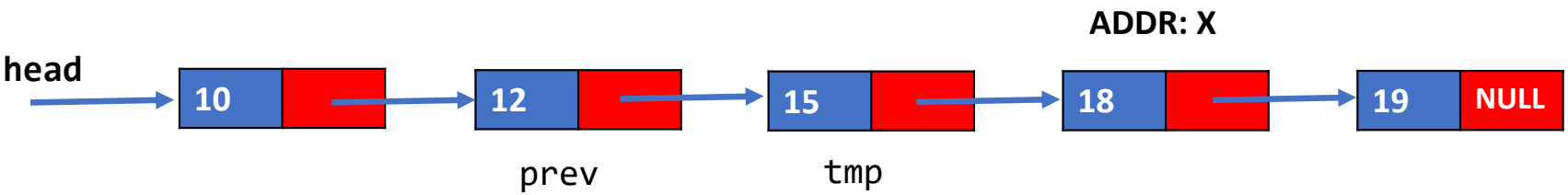
- After deletion



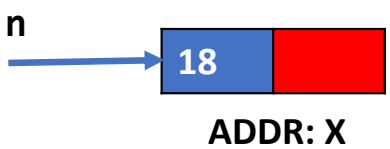
Delete a given node



Delete a given node

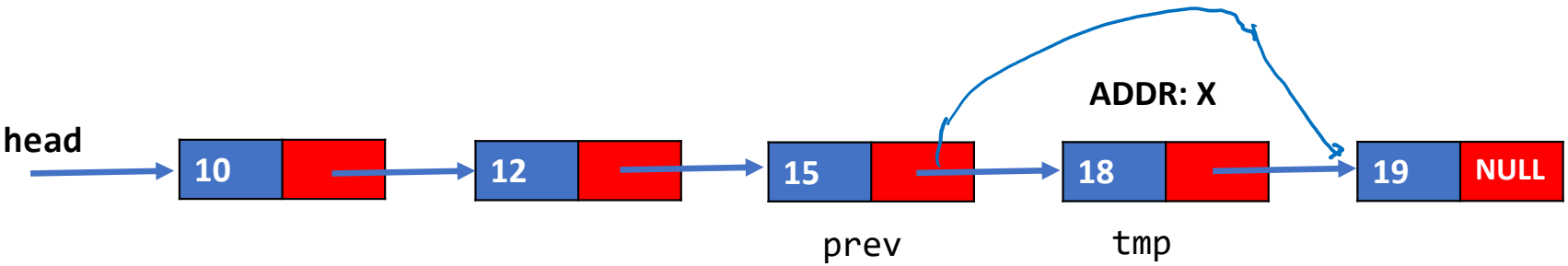



```
prev = head
tmp = head->next
prev = head
tmp = head->next
```

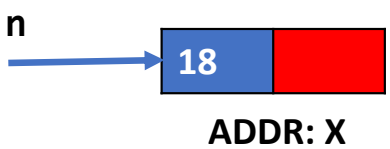


Delete a given node

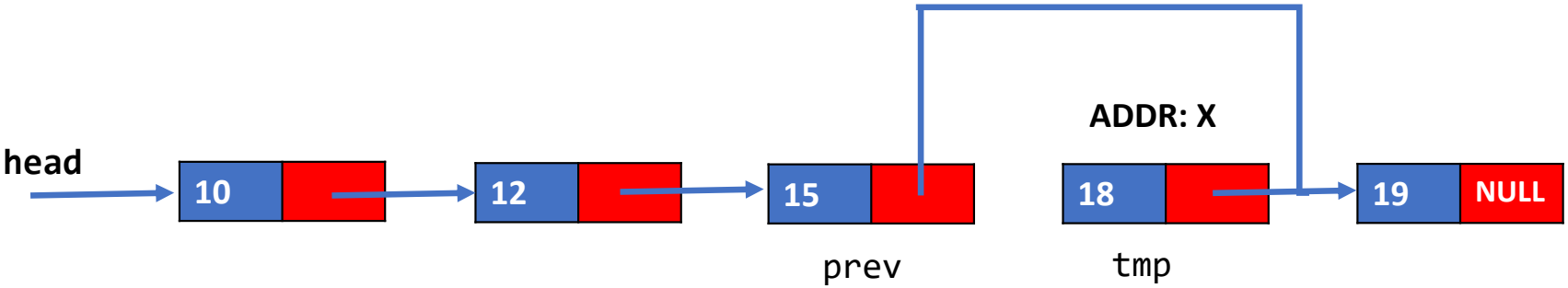
prev->next = tmp->next



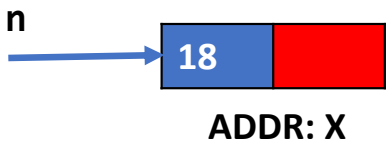

```
prev = head
tmp = head->next
prev = head
tmp = head->next
prev = head
tmp = head->next
```



Delete a given node




```
prev = head
tmp = head->next
prev = head
tmp = head->next
prev = head
tmp = head->next
prev->next = tmp->next
```



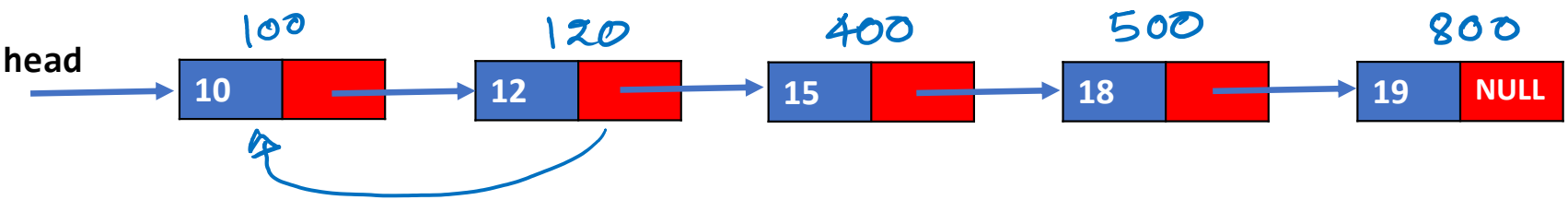
Deleting a given node

```
struct node *delete(struct node **head, struct node *n) {
  struct node *tmp = *head;
  assert(tmp);
  if (tmp == n) {
    *head = tmp->next;  // the given node is the first node
    return tmp;
  }
  if (tmp->next == NULL) {
    return NULL;       // the list has only one element
  }

  struct node *prev = tmp;
  tmp = tmp->next;
  // iterate until we reach the given node
  while (tmp->next != NULL && tmp != n) {
    prev = tmp;
    tmp = tmp->next;
  }
  if (tmp == n) {
    prev->next = tmp->next;  // tmp is the given node
    return tmp;
  }
  return NULL;
}
```

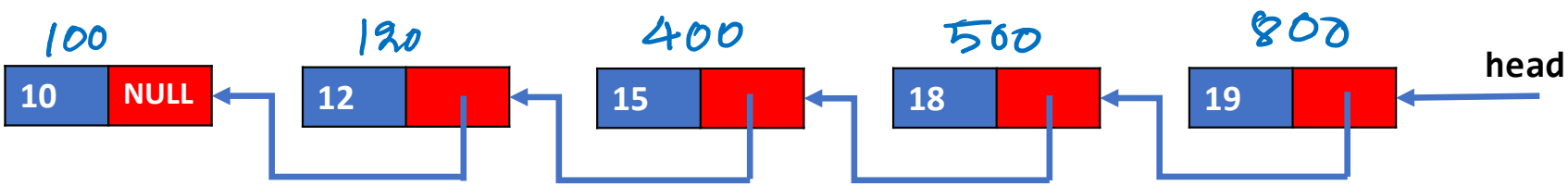
Reversing a linked list

tmp = head
tmp->next = tmp

- Before reversing



- After reversing



Today’s topics

- Circular linked lists
- Stacks

Quiz-2

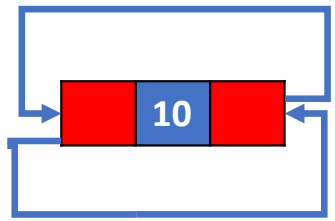
- Closed book examination
- In your answers, you can write pseudo-code or C code
- Pseudo-code should be similar to C, except sometimes we write natural language for brevity
- In pseudo-code, the structure accesses need to be the same as C

Circular linked list

Circular linked list

- Circular linked list can be both singly and doubly
- The prev pointer of the first node points to the last node
- The next pointer of the last node points to the first node
- In a doubly circular linked list, we can efficiently reach the last node from the first node and vice-versa in just one step

Doubly circular linked list with one node



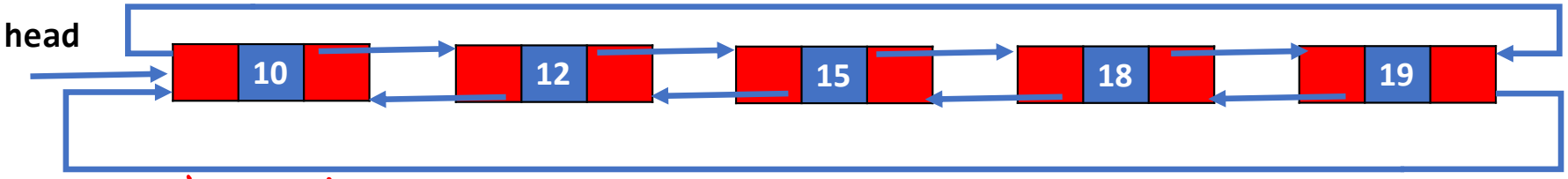

```
struct node {
   int val;
   struct node *next;
   struct node *prev;
};
```

Create:

```
head = NULL;
head = (struct node *) malloc (sizeof (struct node));
assert (head != NULL);
head->val = 10;
head->next = head->prev = head;
```

Printing circular list

```
struct node {
    int val;
    struct node *next;
    struct node *prev;
};
```

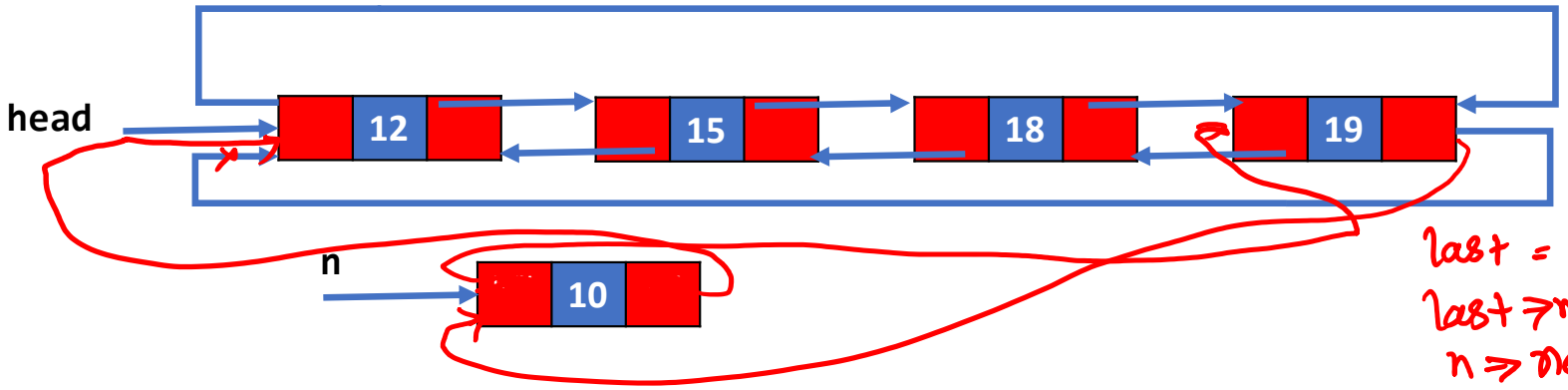



```
tmp = head -> next;
printf ("%d", head -> val);
while ( tmp != head) {
    printf ("%d", tmp -> val);
    tmp = tmp -> next;
}
```

Insert (front)

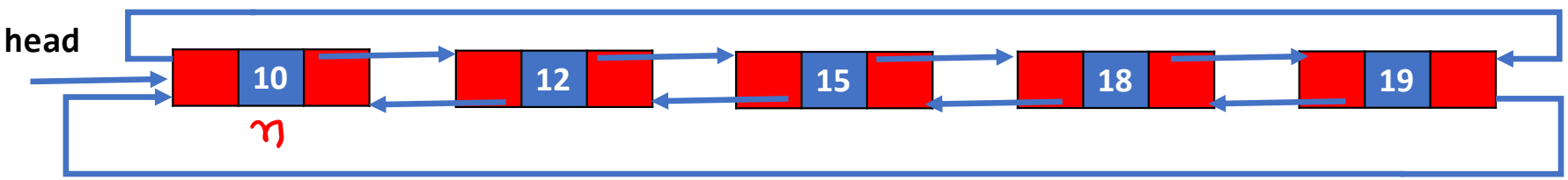
```
struct node {
    int val;
    struct node *next;
    struct node *prev;
};
```

- Before insertion



last = head->prev
last->next = n
n->next = head
n->prev = last
head->prev = n;
head = n;

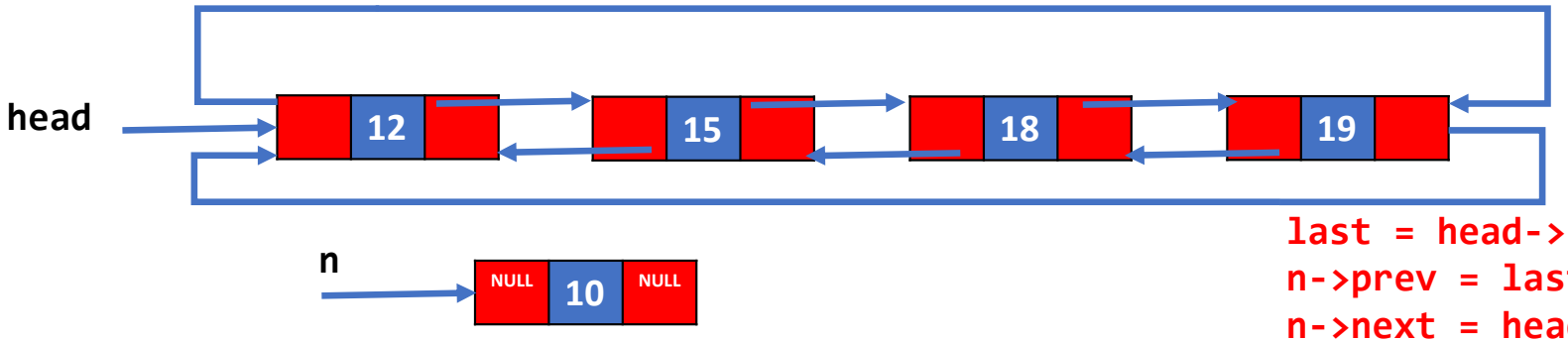
- After insertion



Insert (front)

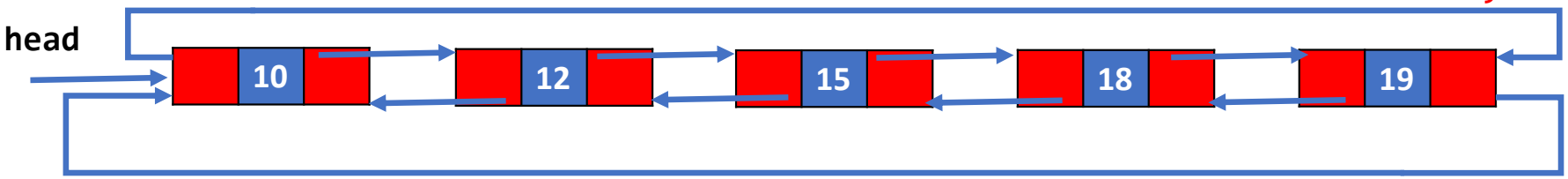
```
struct node {
   int val;
   struct node *next;
   struct node *prev;
};
```

- Before insertion




```
last = head->prev;
n->prev = last;
n->next = head;
head->prev = n;
last->next = n;
head = n;
```

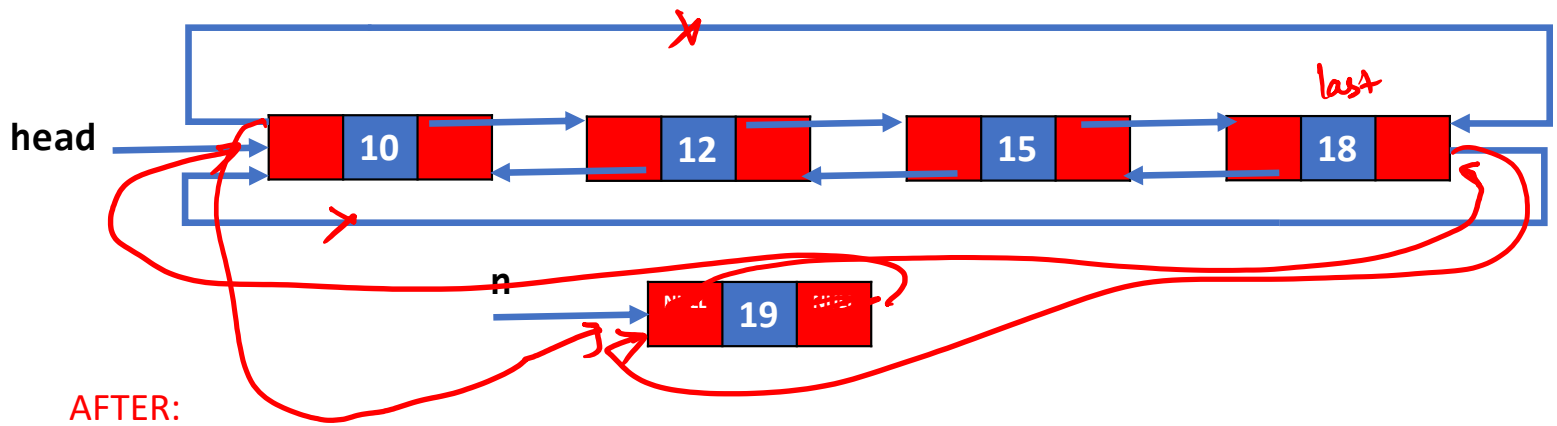
- After insertion



Insert (last)

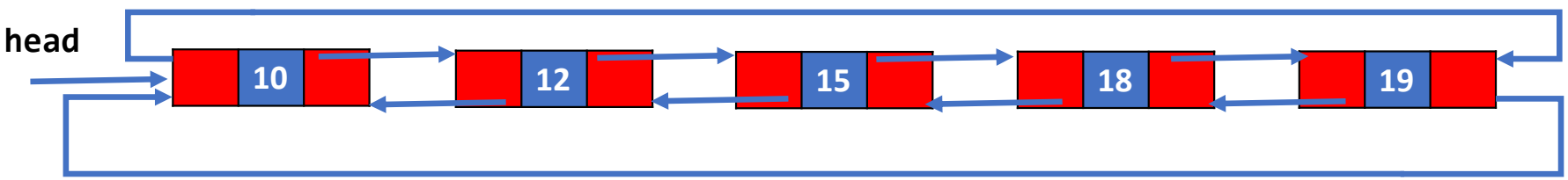
```
struct node {
    int val;
    struct node *next;
    struct node *prev;
};
```

BEFORE:



last = head->prev
n->next = head
n->prev = last
last->next = n
head->prev = n

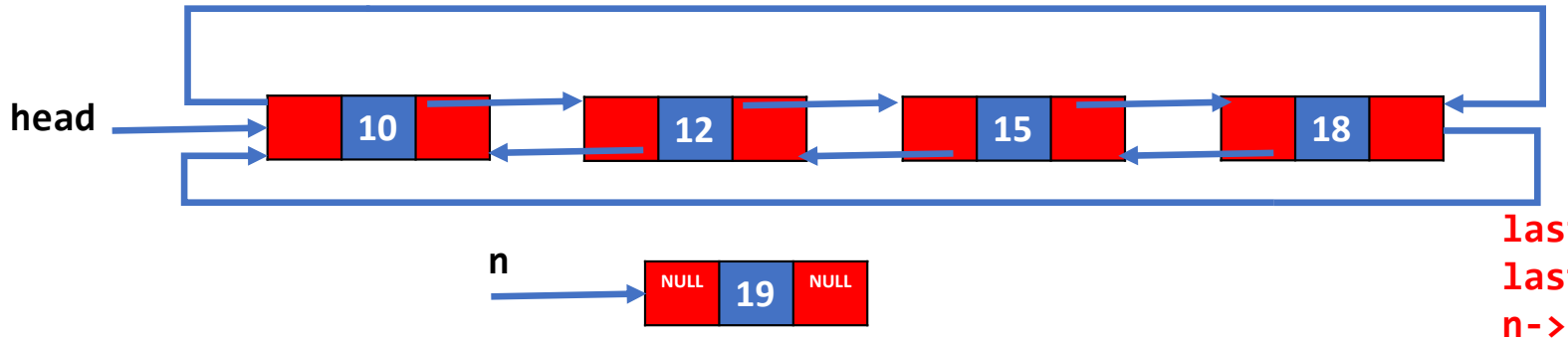
AFTER:



Insert (last)

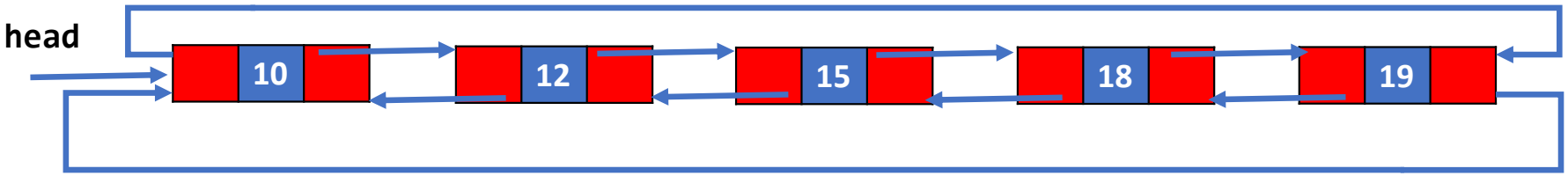
```
struct node {
   int val;
   struct node *next;
   struct node *prev;
};
```

BEFORE:




```
last = head->prev;
last->next = n;
n->prev = last;
n->next = head;
head->prev = n;
```

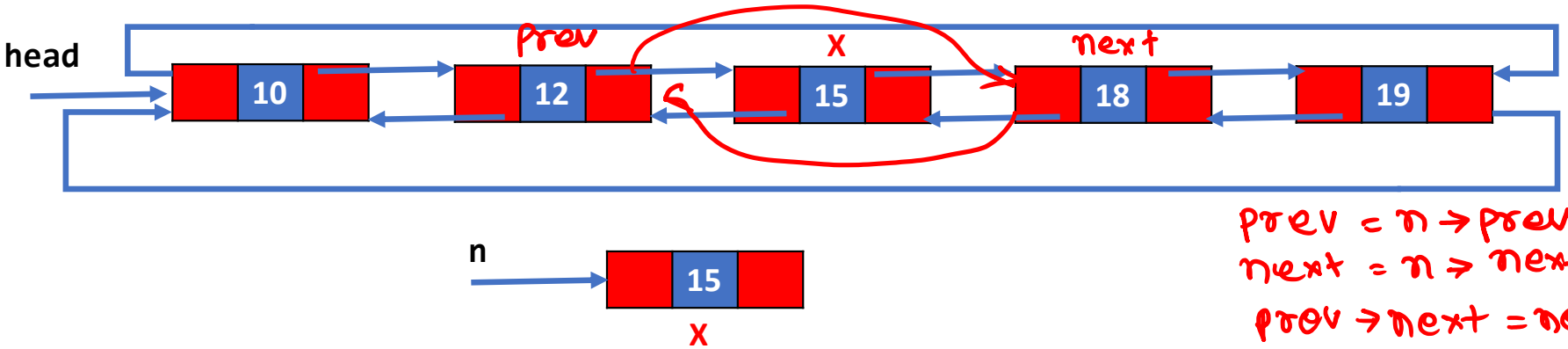
AFTER:



Delete any except first

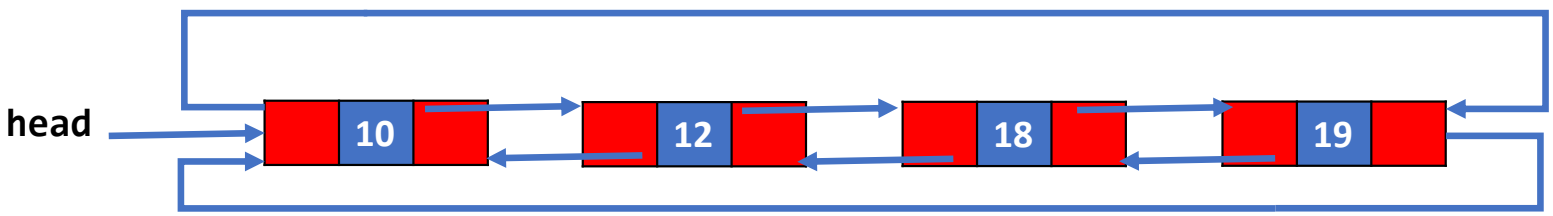
```
struct node {
    int val;
    struct node *next;
    struct node *prev;
};
```

BEFORE:



prev = n -> prev
next = n -> next
prev -> next = next
next -> prev = prev

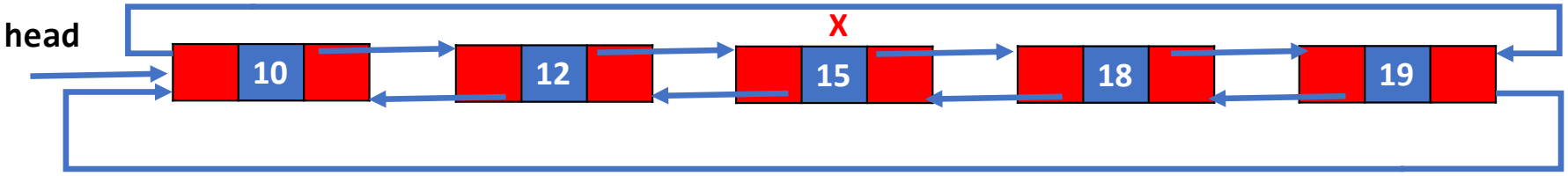
AFTER:

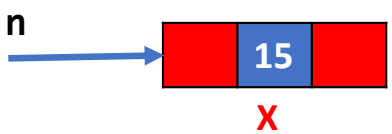


Delete any except first

```
struct node {
   int val;
   struct node *next;
   struct node *prev;
};
```

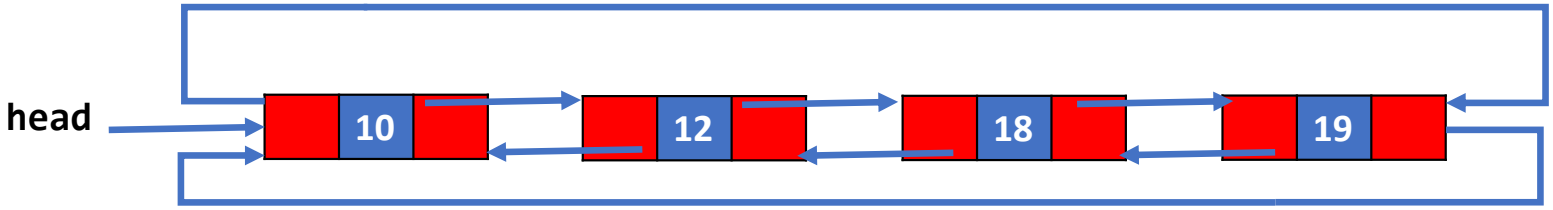
BEFORE:






```
next = n->next
prev = n->prev
prev->next = next
next->prev = prev
```

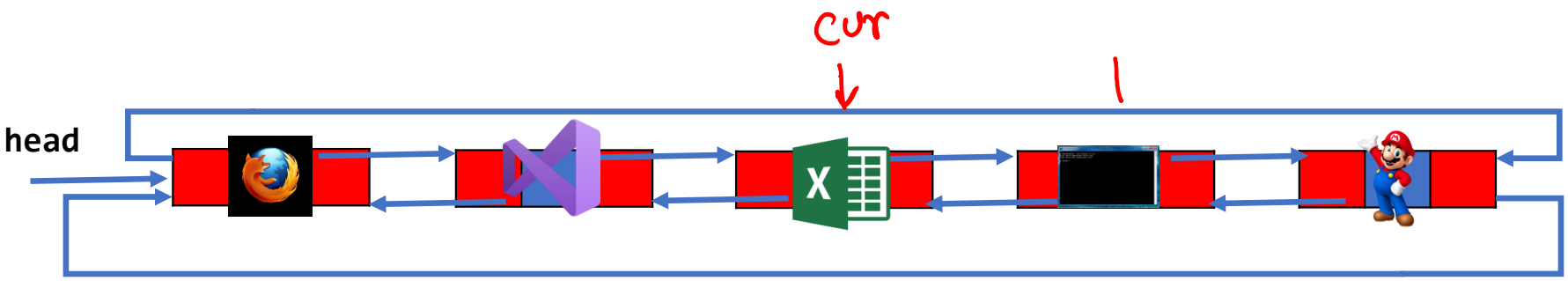
AFTER:



Circular linked list

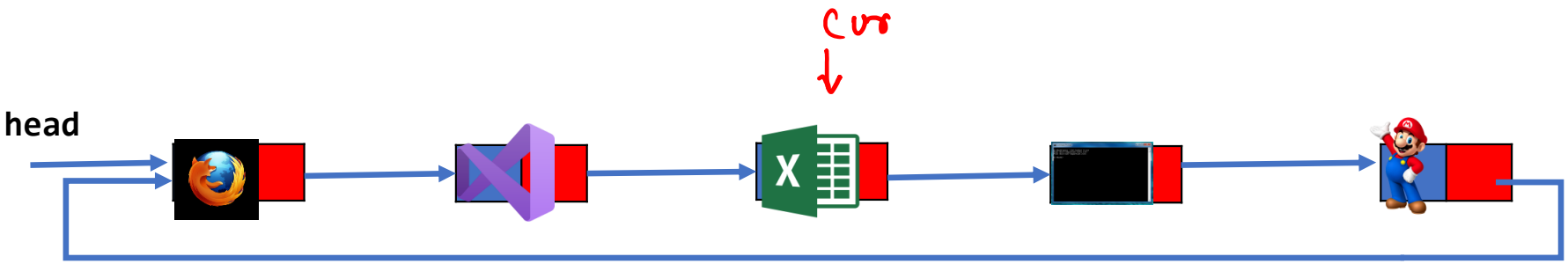
- You can try to implement other operations yourself that we did for the singly linked list

Management of active applications



This Photo by Unknown Author is licensed under CC BY-SA

Management of active applications



Can we implement the applications' list using a circular singly linked list?

This Photo by Unknown Author is licensed under CC BY-SA

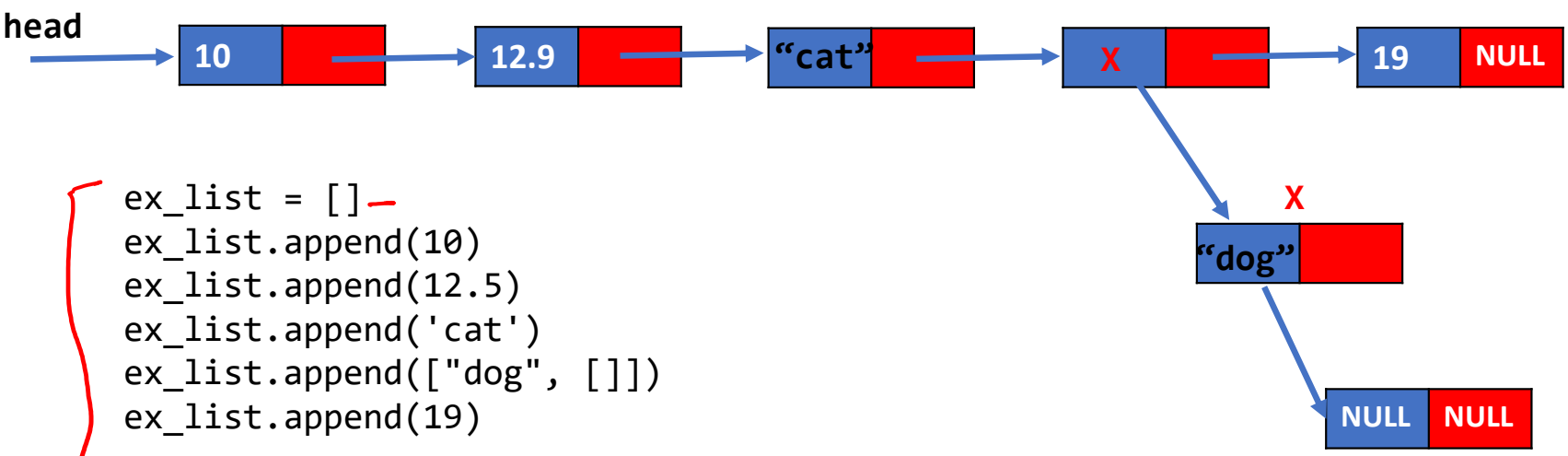
Other application

- Playing a list of songs in a loop

Lists

- In our discussion so far, all the values in the list are of the same type
- How can we support values of different types in the list?

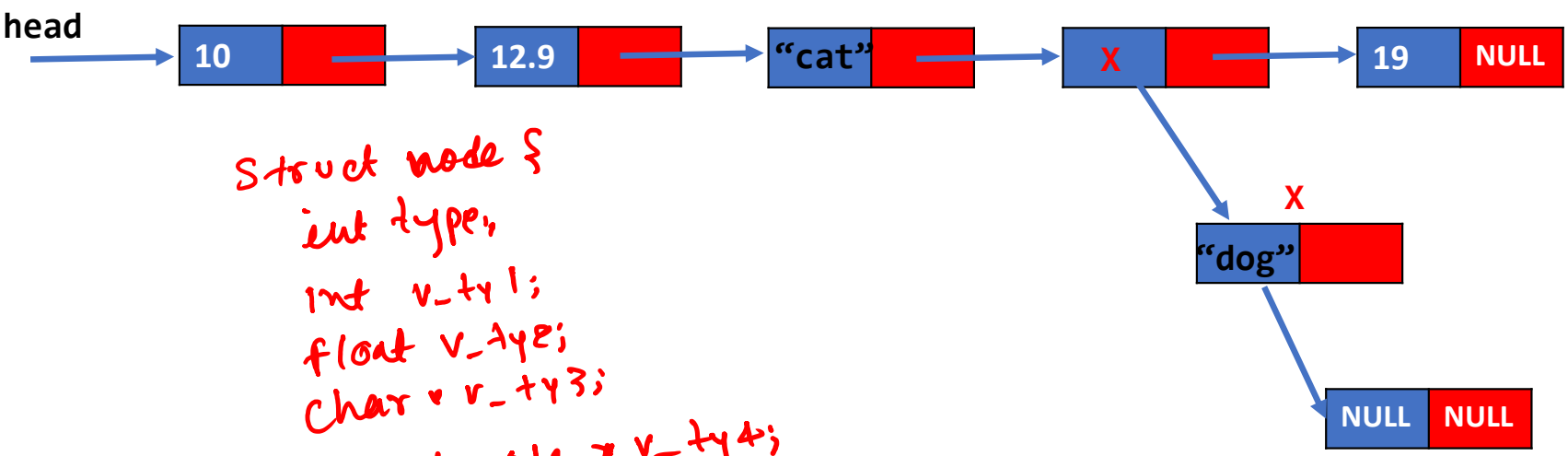
List with values of different type




```
ex_list = []
ex_list.append(10)
ex_list.append(12.5)
ex_list.append('cat')
ex_list.append(["dog", []])
ex_list.append(19)

v = ex_list[2] / 1
print(v)
```

List with values of different type

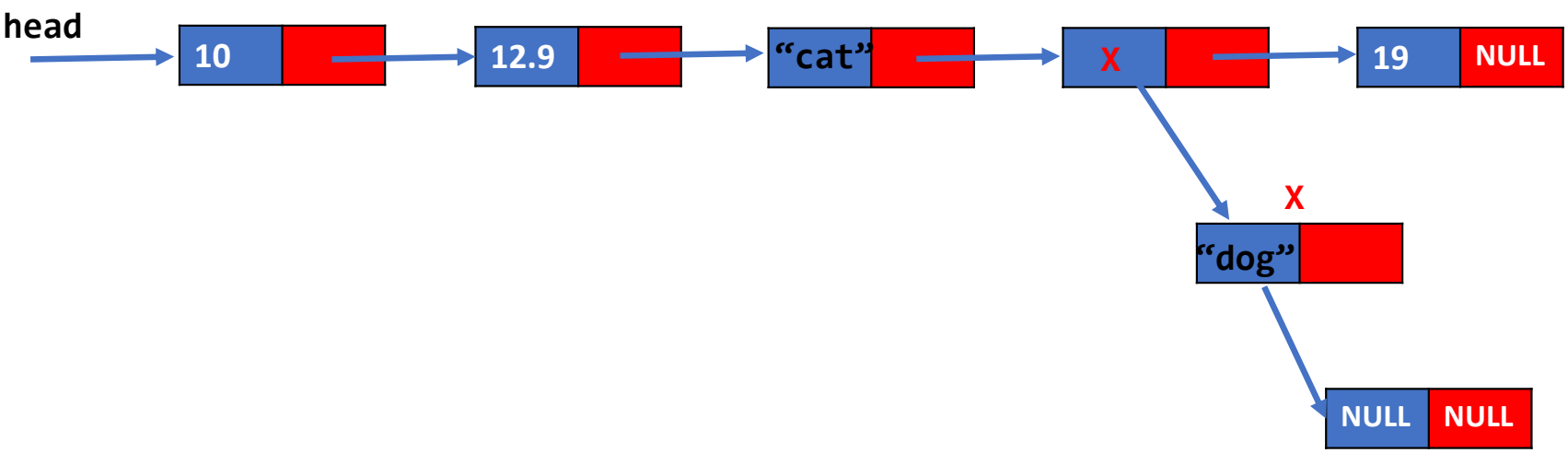
```
struct node {
    int val;
    struct node *next;
};
```




```
struct node {
    int type;
    int v_ty1;
    float v_ty2;
    char * v_ty3;
    struct node * v_ty4;
    struct node * next;
};
```

List with values of different type

```
struct node {
    int type;
    int vty1;
    float vty2;
    char *vty3;
    struct node *vty4;
    struct node *next;
};
```



Exercise

- Write an algorithm to find the middle element of a singly linked list
- Write an algorithm to check if two lists are identical
- Write an algorithm to reverse a doubly linked list recursively
- Write an algorithm to reverse a doubly linked list iteratively
- Write an algorithm to remove duplicates from an unsorted singly linked list
- Write an algorithm to sort a doubly linked list

Stack

References

- Read section-10.1 from Cormem et al.
- Read section-5.1 from Goodrich et al.

Stack

- A stack is the collection of items
- Insertion and deletion of items at the same end called the top of the stack
- Implements last-in, first-out strategy

Stack of plates

This Photo by Unknown Author is licensed under CC BY-SA

Stack ADT

- STACK-EMPTY(S) => returns true if the stack S is empty
- PUSH(S, x) => insert an item x on the top of the stack S
- POP(S) => remove and return the item at the top of the stack S
- TOP(S) => return the item at the top of the stack S without removing

- ADT hides the actual implementation from the clients
  - Internally, a stack can be implemented using a linked list or an array

Example

Operations	Output	Stack Contents
push(5)	–	(5)
push(3)	–	(5 3)
pop()	3	(5)
push(7)	–	(5 7)
pop()	7	(5)
top()	5	(5)
pop()	5	()
pop()	"underflow"	()
top()	"underflow"	()
stack_empty()	True	()
push(9)	–	(9)
push(7)	–	(9 7)
push(7)	–	(9 7 7)
pop()	7	(9 7)
stack_empty()	False	(9 7)

Stack using a singly linked list

Stack

head = NULL

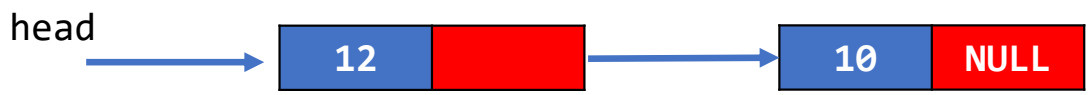
```
Stack is empty.
```

Push

head points to the top of the stack.

push(S, 10)

Push



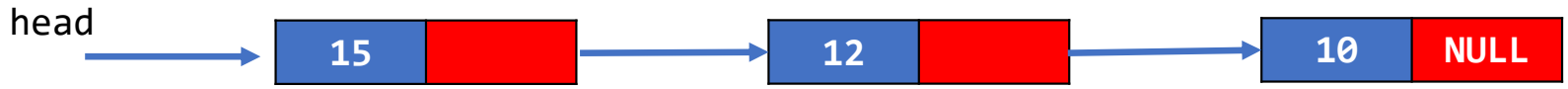
Which list operation is suitable for the push operation.

insert_front

push(S, 10)

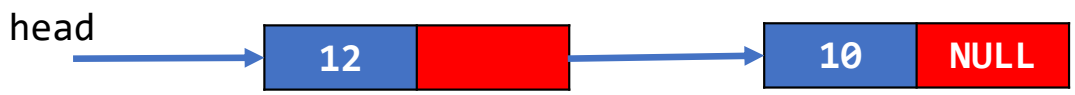
push(S, 12)

Push




```
push(S, 10)
push(S, 12)
push(S, 15)
```

Pop

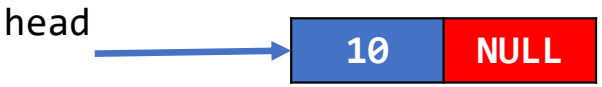


Which list operation is suitable for the pop operation.

delete_front

```
v = pop(S);    // 15
```

Pop




```
v = pop(S);    // 15
v = pop(S);    // 12
```

Pop

head = NULL

```
v = pop(S);   // 15
v = pop(S);   // 12
v = pop(S);   // 10
```

Pop

head = NULL

The pop operation on an empty stack causes underflow.

```
v = pop(S);    // 15
v = pop(S);    // 12
v = pop(S);    // 10
v = pop(S);    // underflow
```

Type of stack

```
struct stack {
    struct node *head;
};
```

Initialize stack

```
struct stack {
    struct node *head;
};

// returns a handle to stack
// all stack operations APIs take this handle as input
struct stack* initialize_stack() {
  struct stack *S = (struct stack*)malloc(sizeof(struct stack));
  assert(S);
  S->head = NULL;
  return S;
}
```

Stack-empty

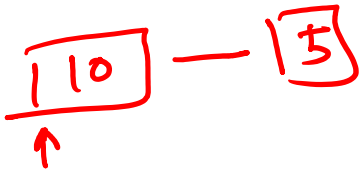
```
// returns 1 if stack is empty
// returns 0 if stack is not empty
int stack_empty(struct stack *S) {
   if (S->head == NULL) {
      return 1;
   }
   return 0;
}
```

Push

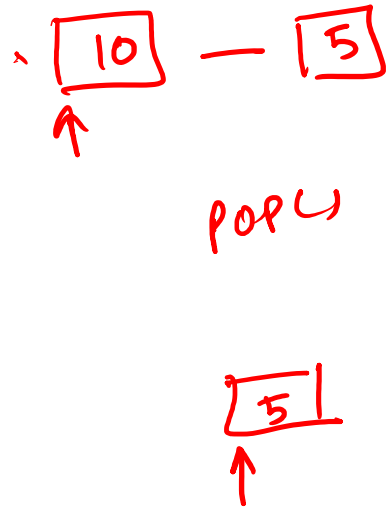
```
// allocate a new node with value val
// if the list is empty the new node is the head
// otherwise, insert the new node at the
// front of the linked list

void push(struct stack *S, int val) {
  struct node *n = allocate_node(val);
  if (S->head == NULL) {
    S->head = n;
  }
  else {
    S->head = insert_front(S->head, n);
  }
}
```

push (5)

push (10)



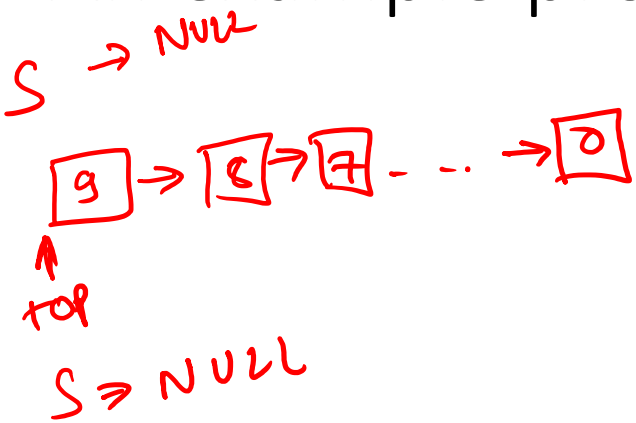
Pop




```
// delete the first node of the linked list and
// return its value

int pop(struct stack *S) {
  if (stack_empty(S)) {
     printf("stack underflow\n");
     exit(0);
  }
  struct node *ret = delete_front(&S->head);
  int retval = ret->val;
  free(ret);
  return retval;
}
```

~~free(ret);~~
~~return ret->val;~~

An example program




```
int main()
{
  struct stack *S = initialize_stack();
  int i;

  assert(stack_empty(S));
  for (i = 0; i < 10; i++) {
    push(S, i);
    print_stack(S);
  }
  assert(!stack_empty(S));
  for (i = 0; i < 10; i++) {
    pop(S);
    print_stack(S);
  }
  assert(stack_empty(S));
  uninitialize_stack(S);
  return 0;
}
```

The client is not expected to access the fields in S (returned from an stack ADT API).

The handle S is invalid after uninitialized_stack.

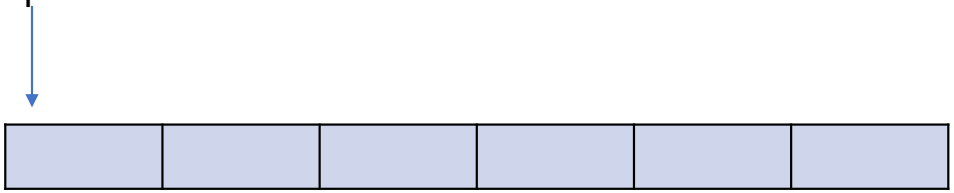
Exercise

- Try to implement the `top(S)` API yourself
- Extend the stack ADT with a new API `size(S)`, which returns the number of elements in the stack
- Implement the `uninitialized_stack(S)` and `print_stack(S)` APIs from the previous slide

Stack using a fixed size array

Stack

top = 0

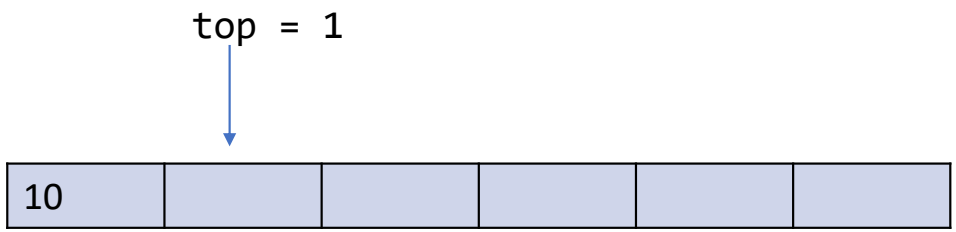


Stack is empty.
Stack size = 6

stack_empty:

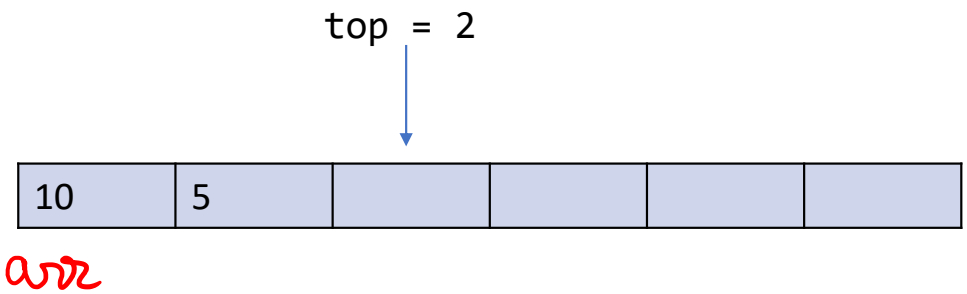
top == 0

Stack



push(10)

Stack

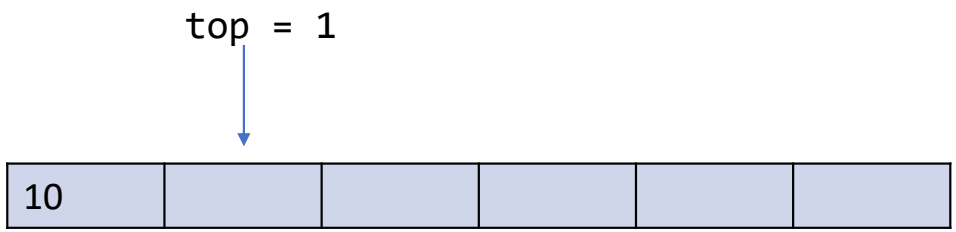



```
push(10)
push(5)
```

push: arr , val, top

arr[top] = val
top = top + 1

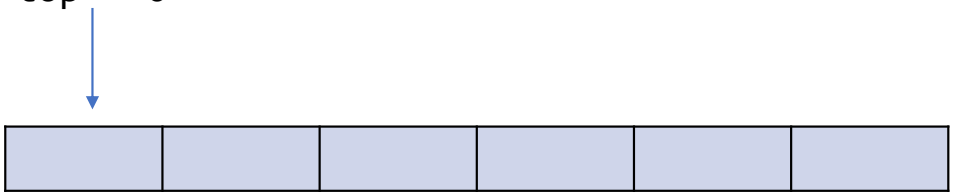
Stack




```
push(10)
push(5)
pop() = 5
```

Stack

top = 0

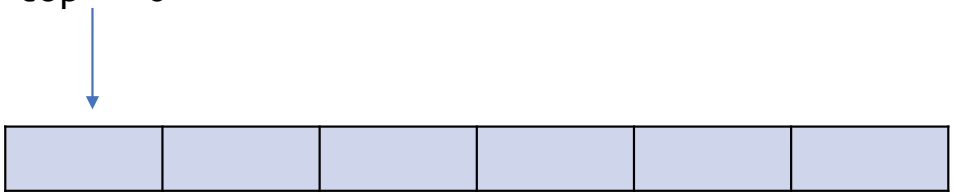


pop: arr, top,
top = top - 1
return arr[top]

push(10)
push(5)
pop() = 5
pop() = 10

Stack

top = 0



```
pop_with_underflow:
    if (top == 0) {
        printf ("underflow");
        exit (0);
    }
    top = top - 1;
    return arr[top];
```

```
push(10)
push(5)
pop() = 5
pop() = 10
pop() "underflow"
```

Stack

top = 6

3	8	5	9	1	2

```
stack_full:  top , arr, size

    if (top == size) {
        overflow;
    }
```

```
push(10)
push(5)
pop() = 5
pop() = 10
pop() "underflow"
push(3)
push(8)
push(5)
push(9)
push(1)
push(2)
```

Stack

top = 6

3	8	5	9	1	2

push(10)
push(5)
pop() = 5
pop() = 10
pop() “underflow”
push(3)
push(8)
push(5)
push(9)
push(1)
push(2)
push(9) “overflow”

push_with_overflow:

```
if ( top == size) {
    printf ("overflow");
    exit(0);
}
arr[top] = val; top = top + 1;
```

No overflow in the linked list based implementation!

Type of stack

```
struct stack {
    int top;
    int size;
    int *arr;
};
```

Type of stack

```
struct stack {
  int top;
  int size;
  int *arr;
};
```

Initialization

```
struct stack* initialize_stack(int size) {
  struct stack *S = (struct stack*)malloc(sizeof(struct stack));
  assert(S);
  S->arr = (int*)malloc(size * sizeof(int));
  assert(S->arr);
  S->top = 0;
  S->size = size;
  return S;
}
```

Stack-empty

```
int stack_empty(struct stack *S) {
  if (S->top == 0) {
    return 1;
  }
  return 0;
}
```

Stack-full

```
int stack_full(struct stack *S) {
  if (S->top == S->size) {
    return 1;
  }
  return 0;
}
```

Push

```
void push(struct stack *S, int val) {
  if (stack_full(S)) {
    printf("Stack overflow\n");
    exit(0);
  }
  S->arr[S->top] = val;
  S->top += 1;
}
```

Pop

```
int pop(struct stack *S) {
  if (stack_empty(S)) {
    printf("Stack underflow\n");
    exit(0);
  }
  int ret = S->arr[S->top - 1];
  S->top -= 1;
  return ret;
}
```

Eliminating stack overflow

Stack

top = 6

3	8	5	9	1	2

push:

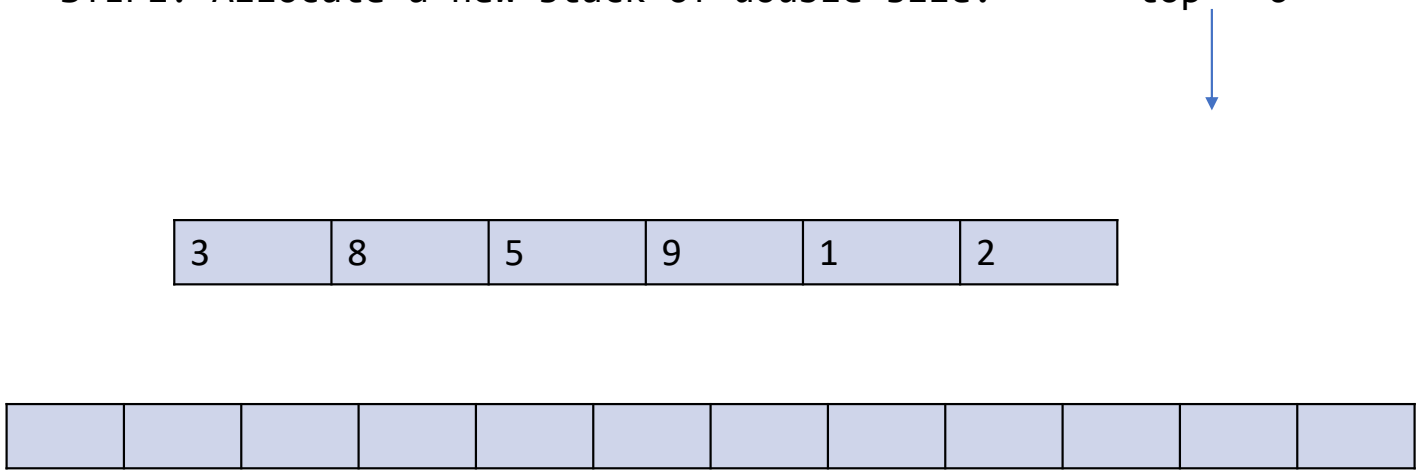
```
push(10)
push(5)
pop() = 5
pop() = 10
pop() “underflow”
push(3)
push(8)
push(5)
push(9)
push(1)
push(2)
push(9) “overflow”
```

No overflow in the linked list based implementation!

Stack

STEP1: Allocate a new stack of double size.

top = 6



No overflow in the linked list based implementation!

```
push(10)
push(5)
pop() = 5
pop() = 10
pop() “underflow”
push(3)
push(8)
push(5)
push(9)
push(1)
push(2)
push(9) “overflow”
```

Stack

STEP1: Allocate a new stack of double size.
STEP2: Copy everything from old to new.

top = 6

3	8	5	9	1	2

top = 6

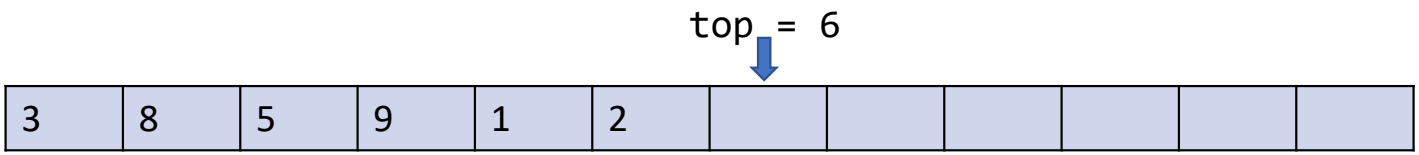
3	8	5	9	1	2						

No overflow in the linked list based implementation!

push(10)
push(5)
pop() = 5
pop() = 10
pop() “underflow”
push(3)
push(8)
push(5)
push(9)
push(1)
push(2)
push(9) “overflow”

Stack

STEP1: Allocate a new stack of double size.
STEP2: Copy everything from old to new.
STEP3: Free the old stack.

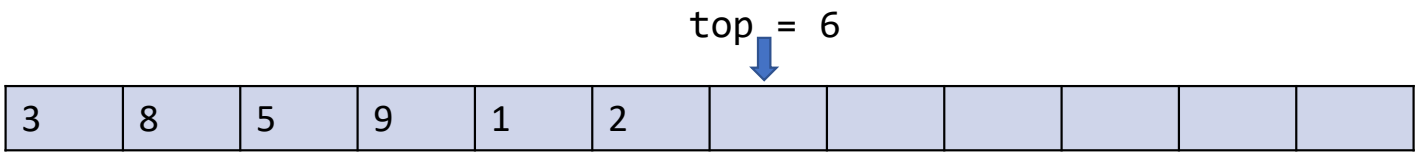


No overflow in the linked list based implementation!

```
push(10)
push(5)
pop() = 5
pop() = 10
pop() “underflow”
push(3)
push(8)
push(5)
push(9)
push(1)
push(2)
push(9) “overflow”
```

Stack

STEP1: Allocate a new stack of double size.
STEP2: Copy everything from old to new.
STEP3: Free the old stack.
STEP4: Update the stack handle to point to the new stack and set the size field to double.



No overflow in the linked list based implementation!

push(10)
push(5)
pop() = 5
pop() = 10
pop() “underflow”
push(3)
push(8)
push(5)
push(9)
push(1)
push(2)
push(9) “overflow”

Stack

STEP1: Allocate new stack of double size.
STEP2: Copy everything from old to new.
STEP3: Free the old stack.
STEP4: Update the stack handle to point to the new stack and set the size field to double.
STEP5: perform the push operation

top = 7

3	8	5	9	1	2	9					

No overflow in the linked list based implementation!

push(10)
push(5)
pop() = 5
pop() = 10
pop() “underflow”
push(3)
push(8)
push(5)
push(9)
push(1)
push(2)
push(9) “no-overflow”

Eliminating stack overflow

- On stack overflow, allocate a new array of double size
- Copy everything from the old array to the new array
- Set the `arr` field in the stack structure to point to the new array
- Double the value of the `size` field in the stack structure
- Why does this work?
  - Why does the rest of the program not hold a reference to the old array?

Eliminating stack overflow

- On stack overflow, allocate a new array of double size
- Copy everything from the old array to the new array
- Set the `arr` field in the stack structure to point to the new array
- Double the value of the `size` field in the stack structure
- Why does this work?
  - Why does the rest of the program not hold a reference to the old array?
    - The clients of an ADT are not expected to access the fields of the stack structure

Push

```
void push(struct stack *S, int val) {
  if (stack_full(S)) {
    // allocate array of double size
    int *arr = (int*)malloc(S->size * 2 * sizeof(int));
    assert(arr);
    // copy from old array to new array
    memcpy(arr, S->arr, S->size * sizeof(int));
    // free the old array
    free(S->arr);
    S->size *= 2;
    S->arr = arr; /* set arr field to new array */
  }
  S->arr[S->top] = val;
  S->top += 1;
}
```

Exercise

- What is the order of the total number of instructions after n push operations in the previous approach?

- Modify the pop algorithm to free half of the buffer when only one-fourth of the buffer is occupied

Today’s topics

- Mid semester feedback
- Linked lists

Mid semester feedback

1. Voice not audible

2. Classes are moving at fast pace, hard to follow

3. Easy things are taught in the class, labs and quizzes are hard

Quiz on 31st May

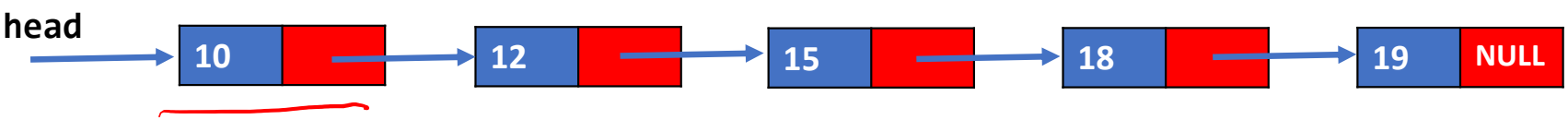
- Closed book examination
- Syllabus: lecture 1 to 10
- The quiz will be conducted during the tutorial slot

Linked lists

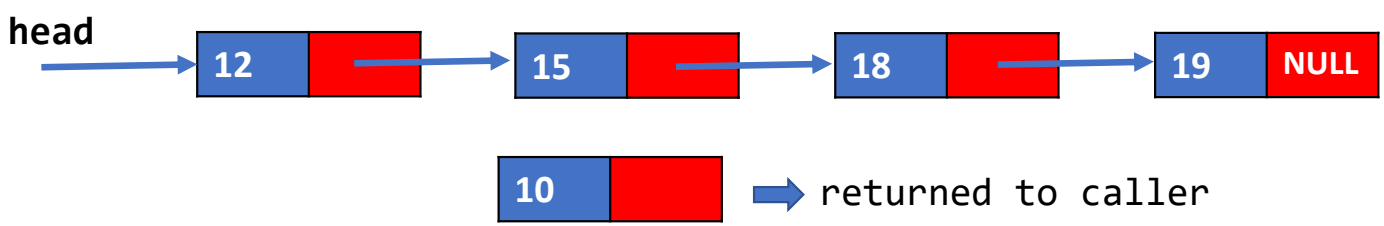
Delete (front)

tmp = head
head = head->next
tmp

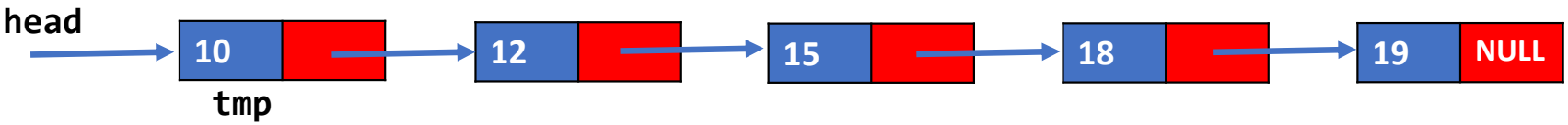
- Before delete



- After delete

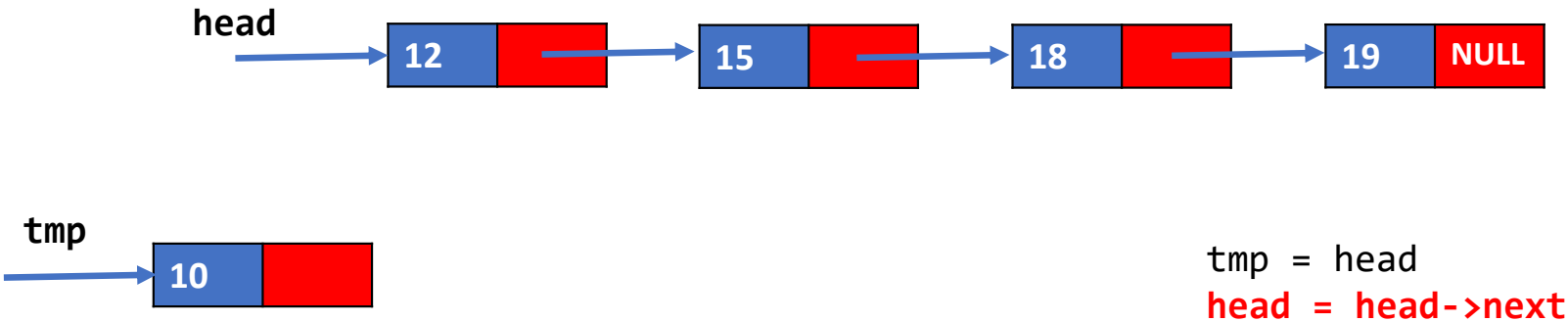


Delete (front)



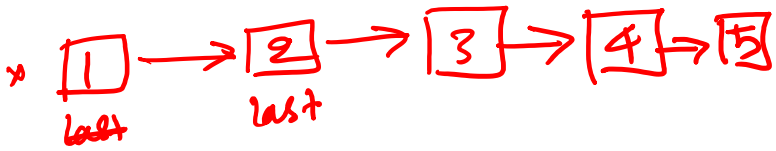

```
tmp = head
```

Delete (front)



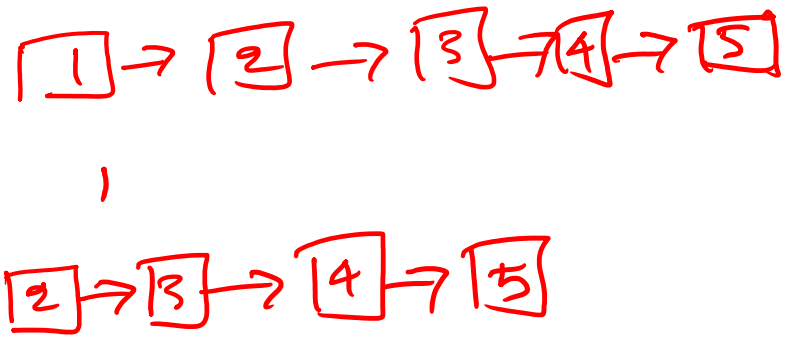

```
tmp = head
head = head->next
```

Delete front (bad)



```
struct node *delete_front(struct node *head) {
  struct node *tmp = head;
  assert(tmp != NULL);
  head = head->next;
  return tmp;
}

int main() {
  struct node *head = allocate_node(1);
  struct node *last = head;
  int i;

  for (i = 2; i <= 5; i++) {
    struct node *n = allocate_node(i);
    last->next = n;
    last = n;
  }
  struct node *tmp = delete_front(head);
  printf("the deleted node is %d\n", tmp->val);

  /* printing values of all nodes in the list */
  while (head != NULL) {
    printf("%d ", head->val);
    head = head->next;
  }
}
```

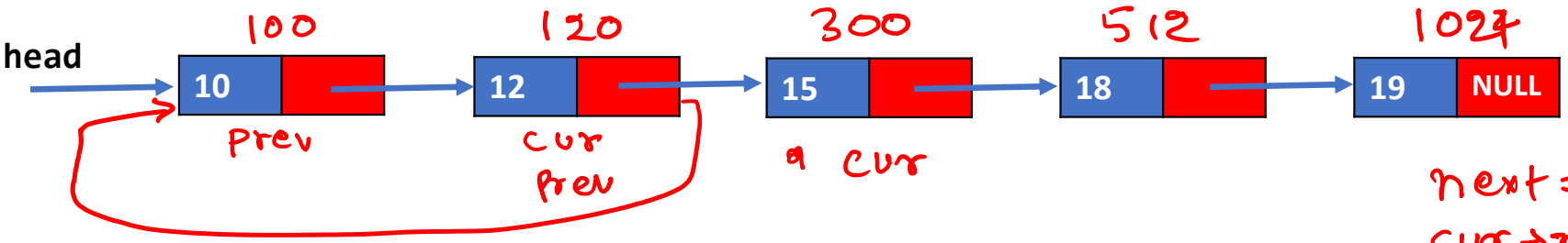
Delete front (good)




```
struct node *delete_front(struct node **head) {
  struct node *tmp = *head;
  assert(tmp != NULL);
  *head = tmp->next;
  return tmp;
}

int main() {
  struct node *head = allocate_node(1);
  struct node *last = head;
  int i;

  for (i = 2; i <= 5; i++) {
    struct node *n = allocate_node(i);
    last->next = n;
    last = n;
  }
  struct node *tmp = delete_front(&head);
  printf(“the deleted node is %d\n”, tmp->val);

  /* printing values of all nodes in the list */
  while (head != NULL) {
    printf("%d ", head->val);
    head = head->next;
  }
}
```

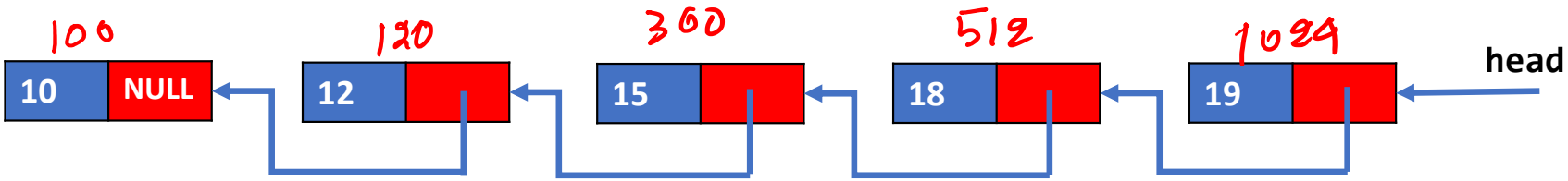
Reversing a linked list

- Before reversing

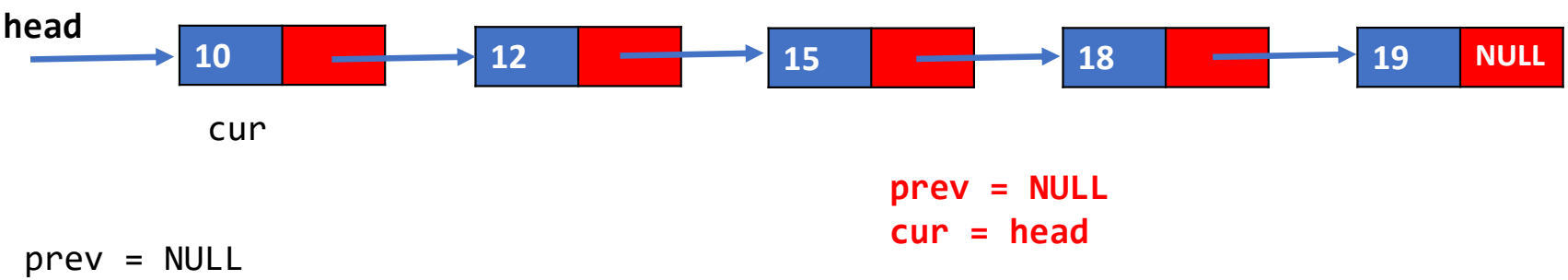


next = cur->next;
cur->next = prev
prev = cur
cur = next

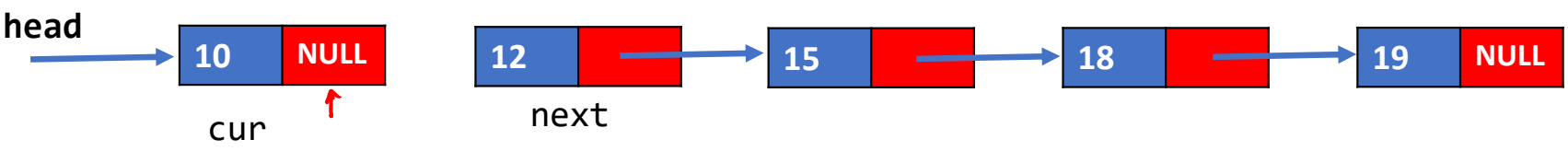
- After reversing



Reversing a linked list



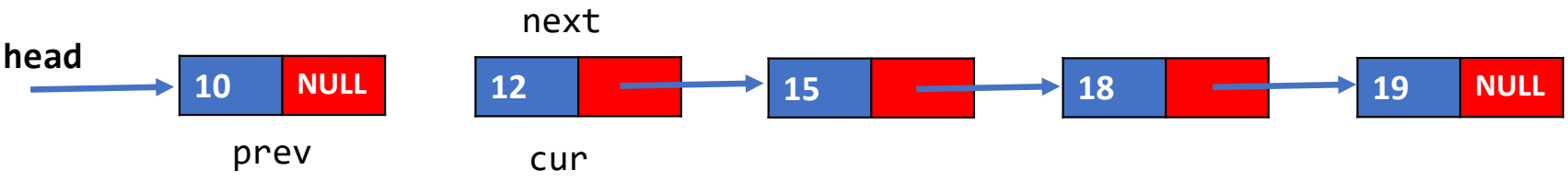
Reversing a linked list




```
prev = NULL
```

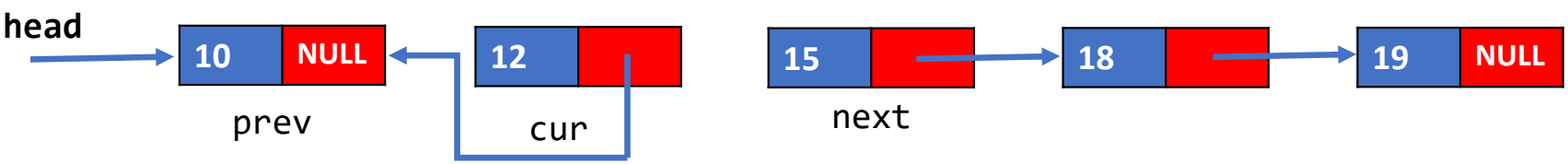
```
prev = NULL
cur = head
next = cur->next
cur->next = prev
```

Reversing a linked list



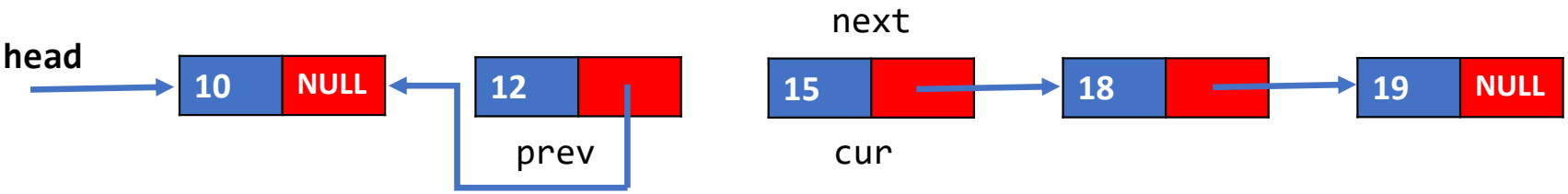

```
prev = NULL
cur = head
next = cur->next
cur->next = prev
prev = cur
cur = next
```

Reversing a linked list



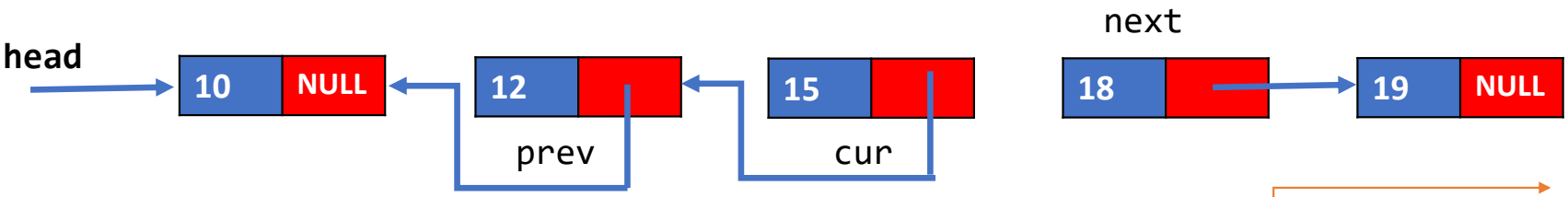

```
prev = NULL
cur = head
next = cur->next
cur->next = prev
prev = cur
cur = next
next = cur->next
cur->next = prev
```

Reversing a linked list




```
prev = NULL
cur = head
next = cur->next
cur->next = prev
prev = cur
cur = next
next = cur->next
cur->next = prev
prev = cur
cur = next
```

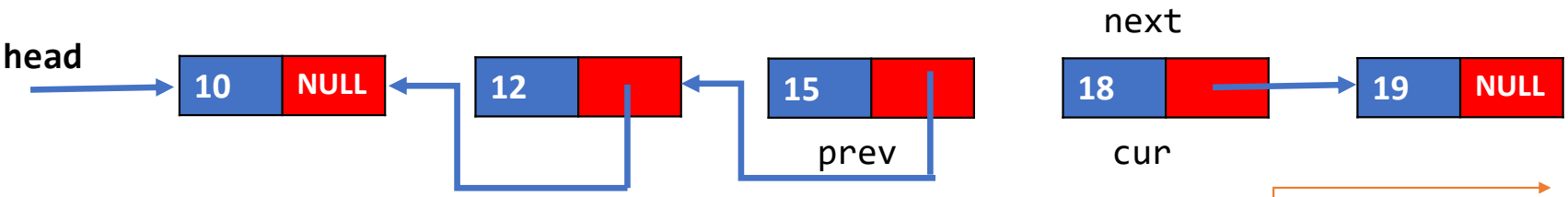
Reversing a linked list




```
prev = NULL
cur = head
next = cur->next
cur->next = prev
prev = cur
cur = next
next = cur->next
cur->next = prev
prev = cur
cur = next
```

```
next = cur->next
cur->next = prev
```

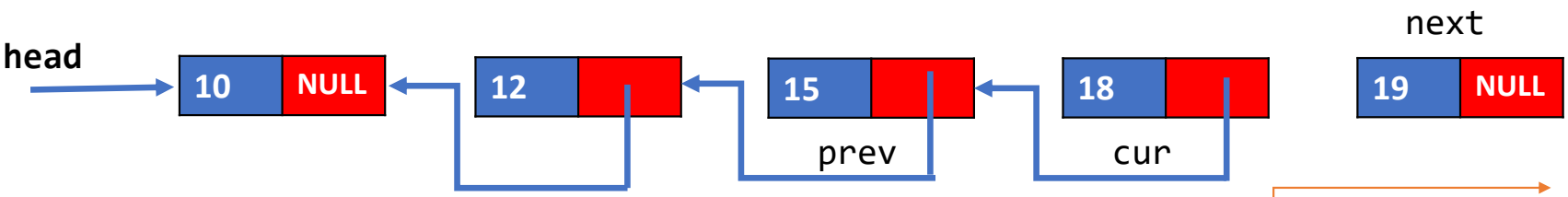
Reversing a linked list




```
prev = NULL
cur = head
next = cur->next
cur->next = prev
prev = cur
cur = next
next = cur->next
cur->next = prev
prev = cur
cur = next
```

```
next = cur->next
cur->next = prev
prev = cur
cur = next
```

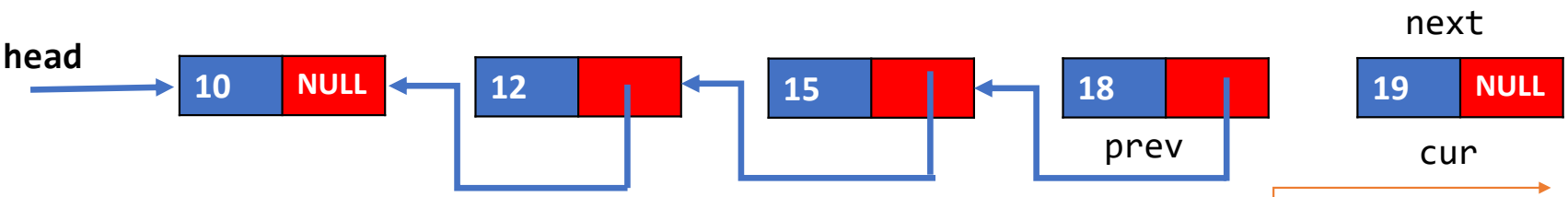
Reversing a linked list




```
prev = NULL
cur = head
next = cur->next
cur->next = prev
prev = cur
cur = next
next = cur->next
cur->next = prev
prev = cur
cur = next
```

```
next = cur->next
cur->next = prev
prev = cur
cur = next
next = cur->next
cur->next = prev
```

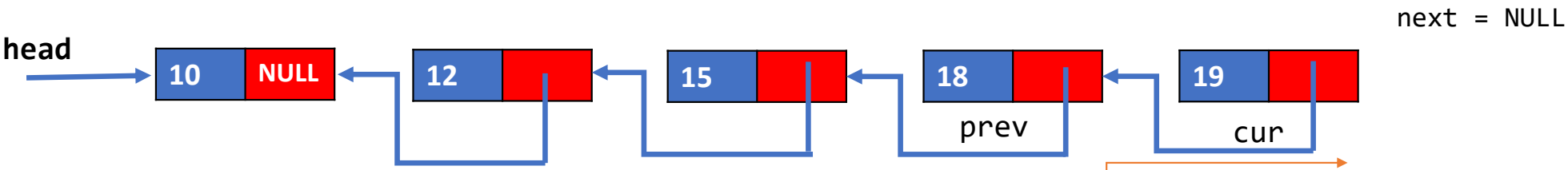
Reversing a linked list




```
prev = NULL
cur = head
next = cur->next
cur->next = prev
prev = cur
cur = next
next = cur->next
cur->next = prev
prev = cur
cur = next
```

```
next = cur->next
cur->next = prev
prev = cur
cur = next
next = cur->next
cur->next = prev
prev = cur
cur = next
```

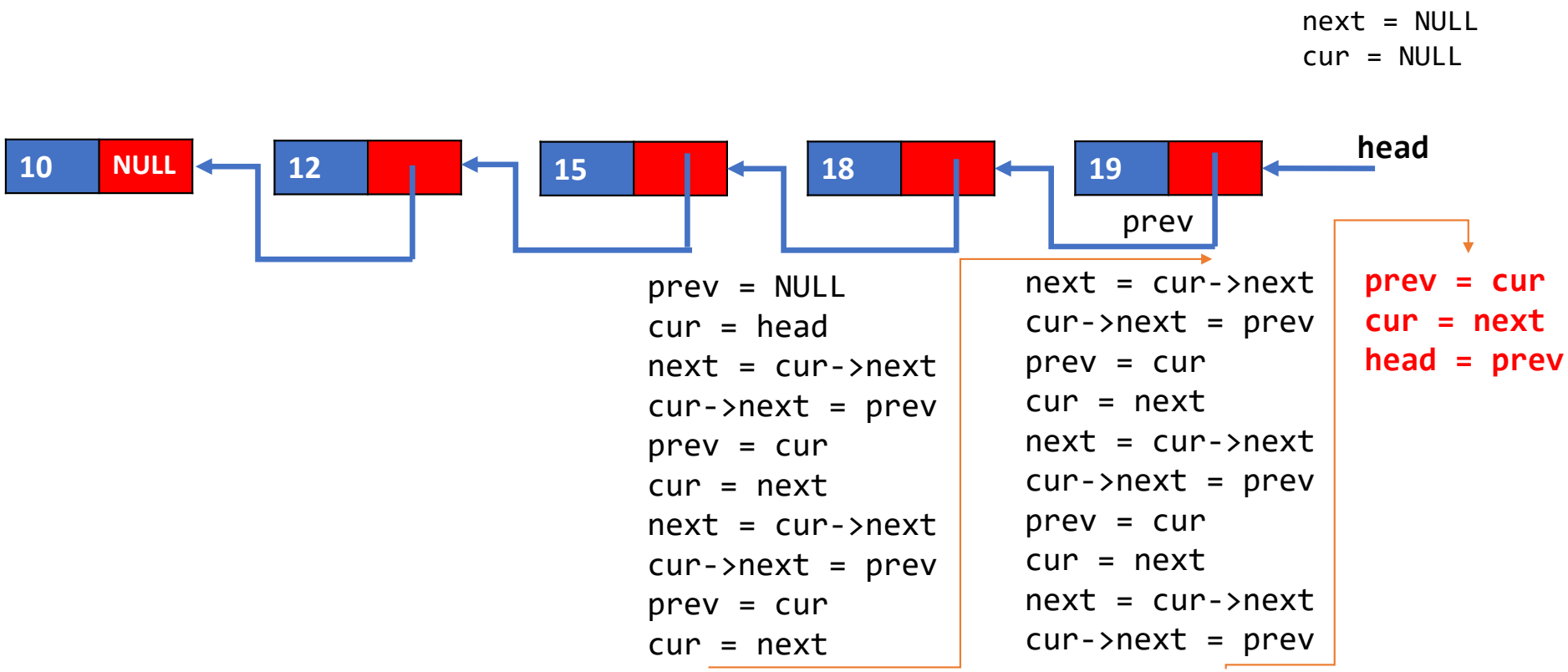
Reversing a linked list




```
prev = NULL
cur = head
next = cur->next
cur->next = prev
prev = cur
cur = next
next = cur->next
cur->next = prev
prev = cur
cur = next
```

```
next = cur->next
cur->next = prev
prev = cur
cur = next
next = cur->next
cur->next = prev
prev = cur
cur = next
next = cur->next
cur->next = prev
```

Reversing a linked list

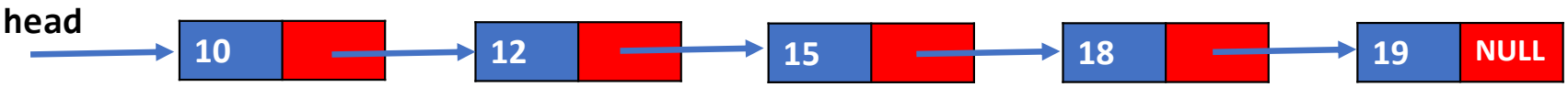


Reversing a linked list

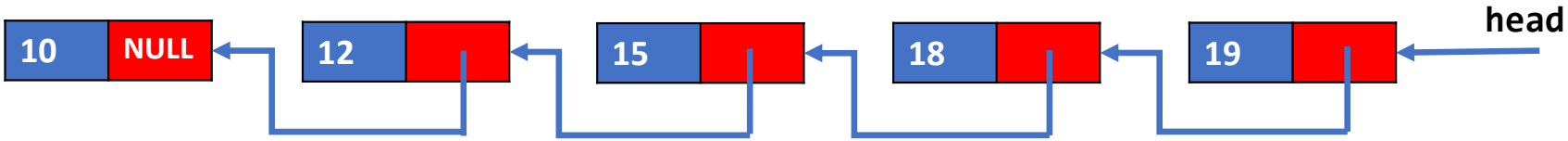
```
struct node *reverse_list(struct node *head) {
  struct node *prev = NULL;
  struct node *cur = head;
  struct node *next;

  while (cur != NULL) {
    next = cur->next;
    cur->next = prev;
    prev = cur;
    cur = next;
  }
  return prev; // returns the new head
}
```

Reversing a linked list (Recursive)

- Before reversing

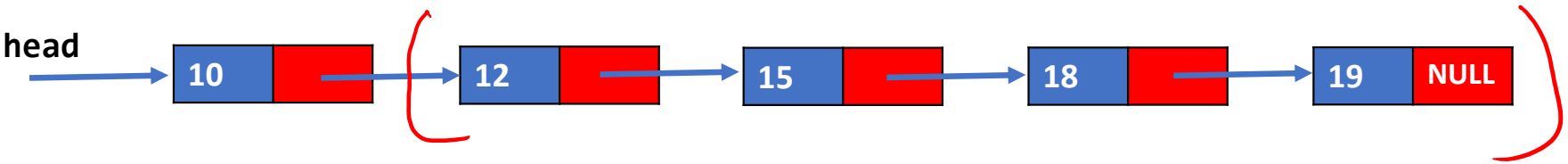


- After reversing

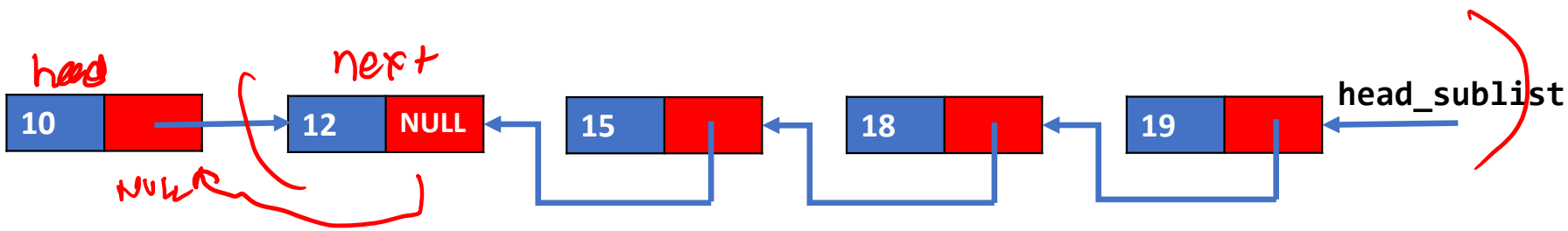


Recursive step

```
reverse_list ( head)
{
    next = head->next;
    new head = reverse_list(head->next)
    next->next = head;
    head->next = NULL;
    new head
```

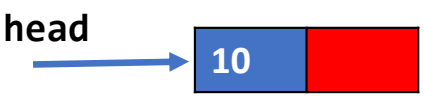


Recursively reverse the rest of the list starting at `head->next`



Base case

- List has just one element



return head

Reversing a linked list (Recursive)

- Base case: if the linked list has just one element, the reverse is the same as the original, return the linked list

- Recursive step:
  - `next = head->next`
  - `head_sublist` = recursively reverse the linked list starting at `next`
  - `next->next = head`
  - `head->next = NULL`
  - `head_sublist` is the head of the reversed list

Reversing a linked list (Recursive)

```
struct node *reverse_list_rec(struct node *head) {
  assert(head != NULL); // how to support NULL list?
  struct node *next = head->next;
  if (next == NULL) {
    return head;
  }
  struct node *head_sublist = reverse_list_rec(next);
  next->next = head;
  head->next = NULL;
  return head_sublist;
}
```

→ if (head == NULL)
return NULL;

```
reverse_list_rec(struct node *head) {
    if (head == NULL)
        return NULL;
    return reverse_list_rec(head);
```

Applications of linked list

Polynomial operations using linked list

- Polynomial can be stored in a linked list

$$5x^3 + 3x^2 + 1$$

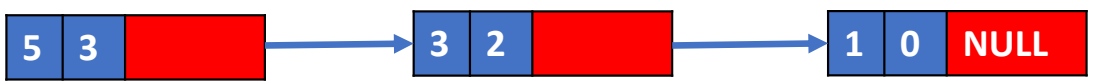
Polynomial operations using linked list

struct node {
int coefficient;
int exponent;
struct node *next
};

- Polynomial can be stored in a linked list

$$5x^3 + 3x^2 + 1$$

We can store coefficients and exponents in the linked list nodes



Type of the linked list node

Type of the linked list node

```
struct poly_node {
   int coefficient;
   int exponent;
   struct poly_node *next;
};
```

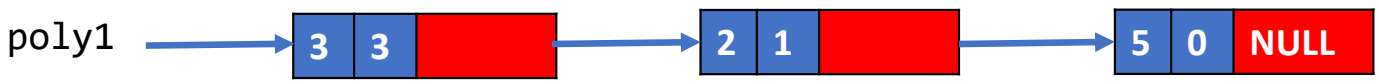
Adding two polynomials

$$3x^3 + 2x + 5$$
$$9x^7 + 3x^5 + 7x^3 + 1$$

Let's say that the linked list nodes are sorted in the descending order of exponent values.

Adding two polynomials

$$3x^3 + 2x + 5$$

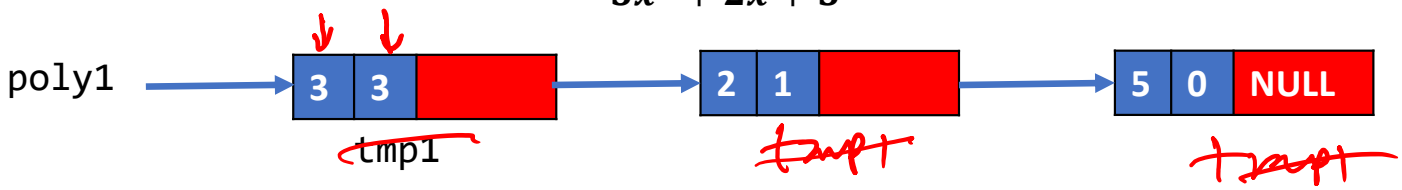


$$9x^7 + 3x^5 + 7x^3 + 1$$

Create a new linked list that represents the sum of both the polynomials.

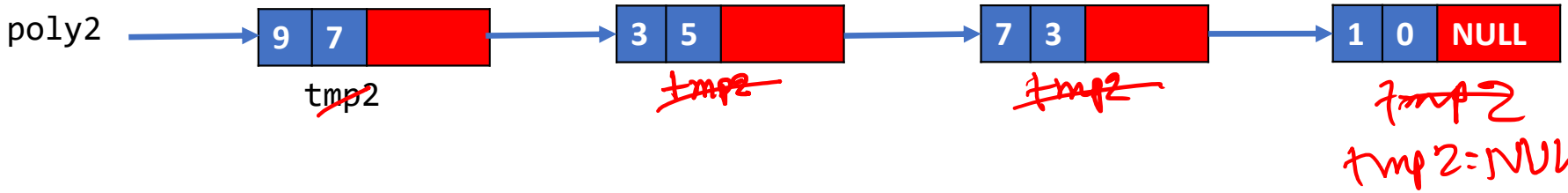
Adding two polynomials

$$3x^3 + 2x + 5$$

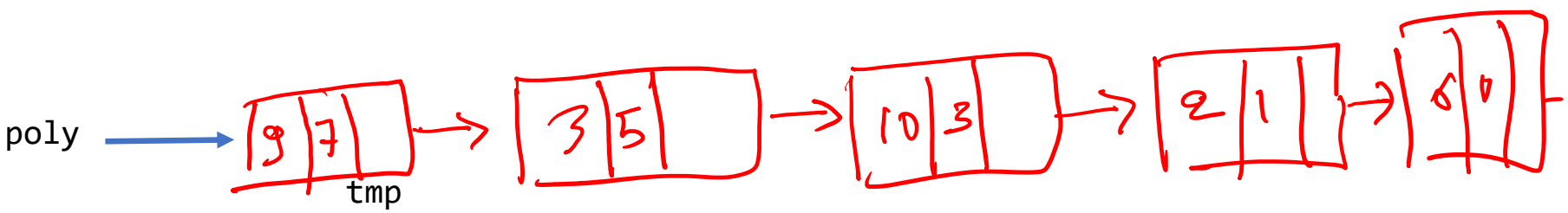


tmp1 = NULL

$$9x^7 + 3x^5 + 7x^3 + 1$$

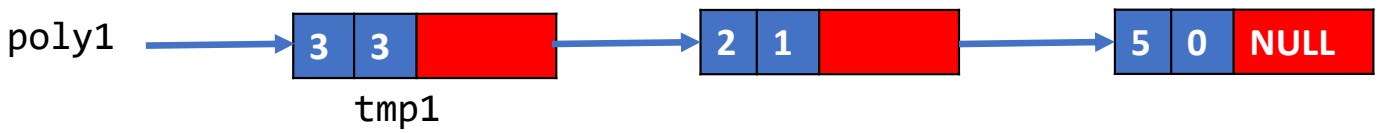


tmp2 = NULL

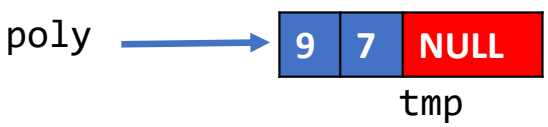


Adding two polynomials

$$3x^3 + 2x + 5$$

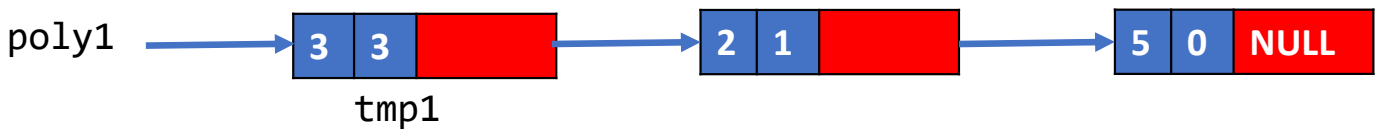


$$9x^7 + 3x^5 + 7x^3 + 1$$

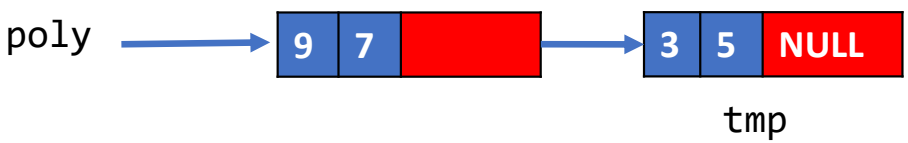


Adding two polynomials

$$3x^3 + 2x + 5$$

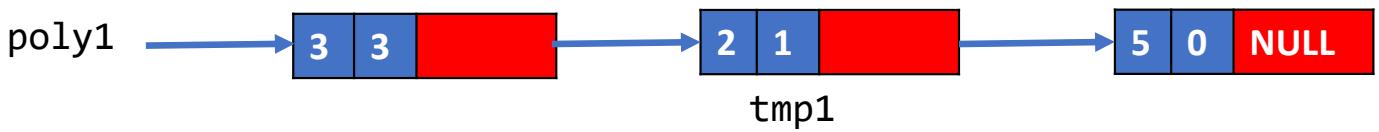


$$9x^7 + 3x^5 + 7x^3 + 1$$

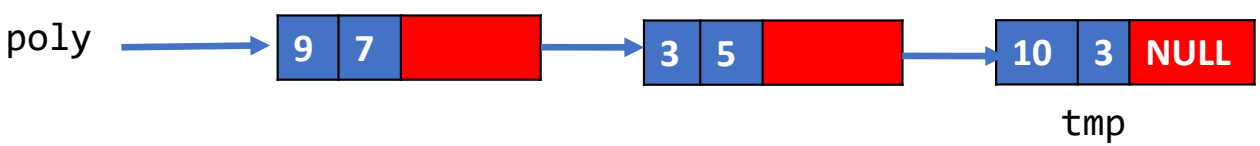


Adding two polynomials

$$3x^3 + 2x + 5$$

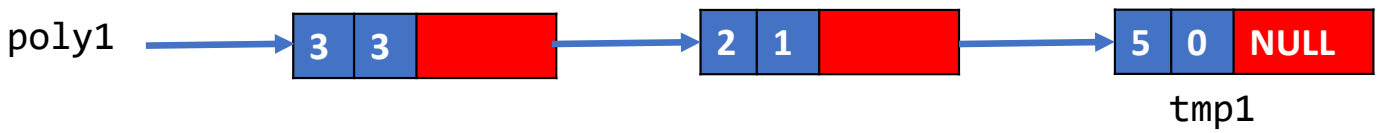


$$9x^7 + 3x^5 + 7x^3 + 1$$

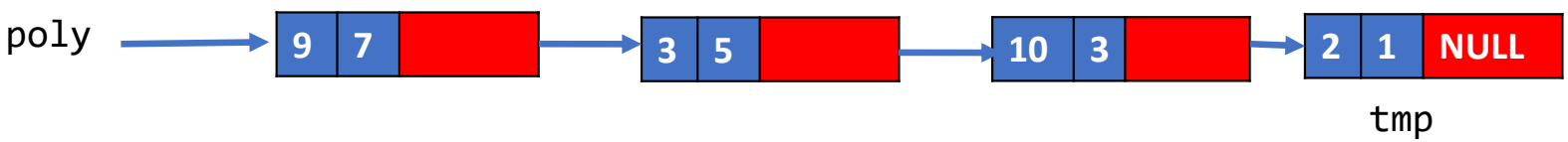


Adding two polynomials

$$3x^3 + 2x + 5$$

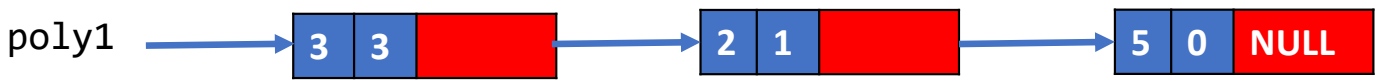


$$9x^7 + 3x^5 + 7x^3 + 1$$



Adding two polynomials

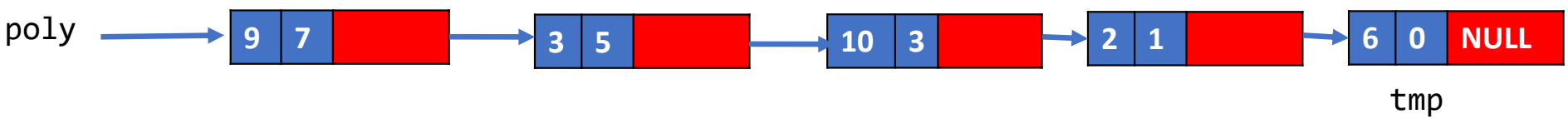
$$3x^3 + 2x + 5$$



tmp1 = NULL
tmp2 = NULL

$$9x^7 + 3x^5 + 7x^3 + 1$$

$$9x^7 + 3x^5 + 10x^3 + 2x + 1$$



Singly linked list

Singly linked list

- The linked list that we discussed so far is also called a singly linked list
- The address of the first node of a singly linked list is stored in a variable head
- Linked list can only be traversed in the forward direction starting from the head
- An empty linked list is represented using NULL
- Every node in a singly linked list contains a reference to the singly linked list that starts from the next node
  - The last node of the singly linked list contains a reference to the empty list

Doubly linked list

Search returns a linked list node

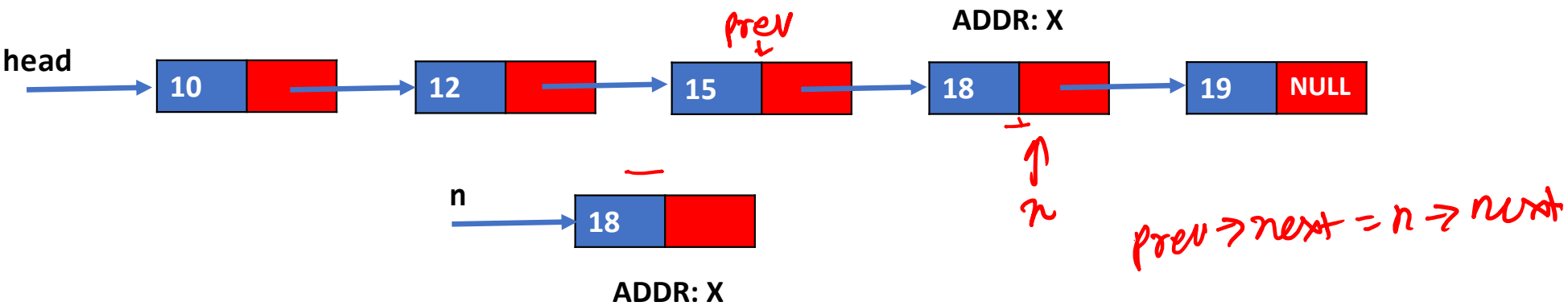
```
// Returns a linked list (starts at head) node that contains
// the input argument val

struct node* search(struct node *head, int val) {
  struct node *tmp = head;
  while (tmp != NULL) {
    if (tmp->val == val) {
      return tmp;
    }
    tmp = tmp->next;
  }
  return NULL;
}
```

Delete a given node

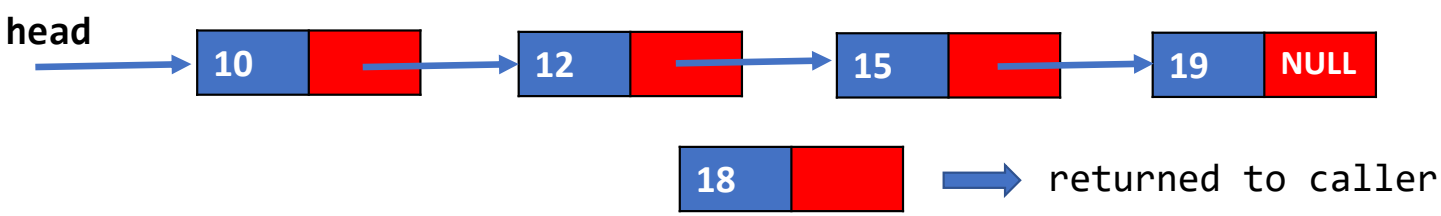
Why do we need to iterate nodes to delete n?

Can we store additional information in linked list nodes to prevent iteration?

- Before deletion



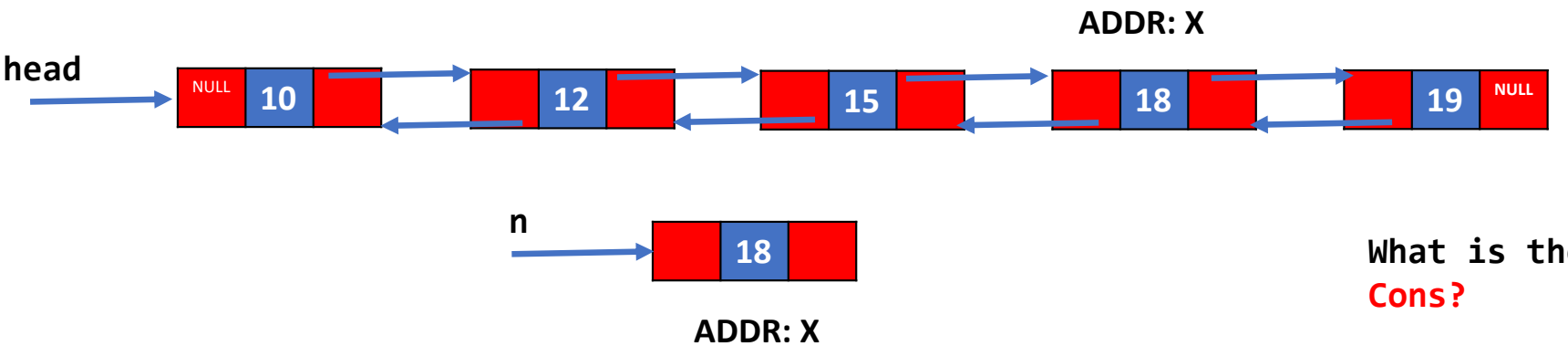
- After deletion



Delete a given node

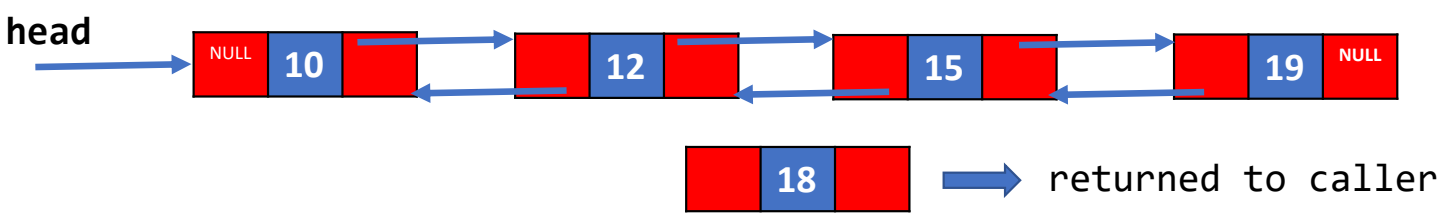
```
struct node {
  int val;
  struct node *next;
  struct node *prev;
};
```

- Before deletion



What is the type?
Cons?

- After deletion



Type

- Doubly linked list node

```
struct node {
    int val;
    struct node *next;
    struct node *prev;
};
```

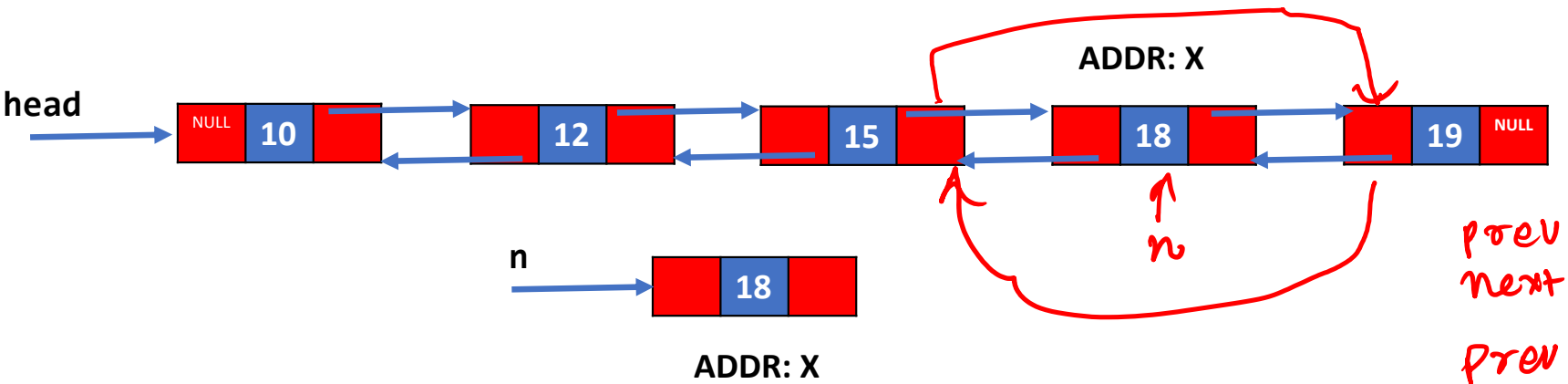
- head

```
struct node*
```

Delete a given node at pos > 0

```
struct node {
    int val;
    struct node *next;
    struct node *prev;
};
```

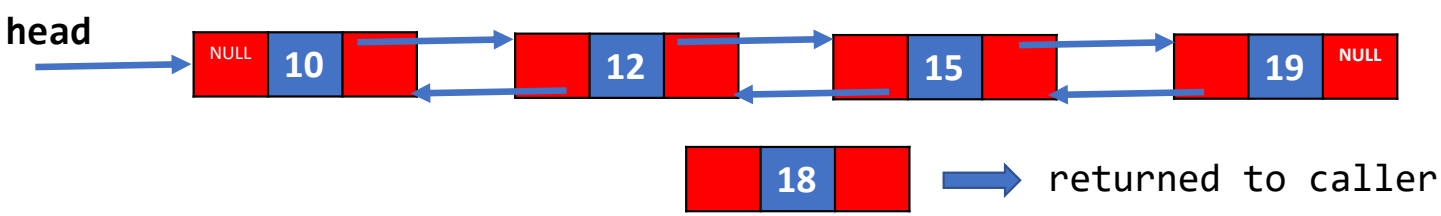
- Before deletion



prev = n -> prev
next = n -> next

prev -> next = next
if (next != NULL)
next -> prev = prev

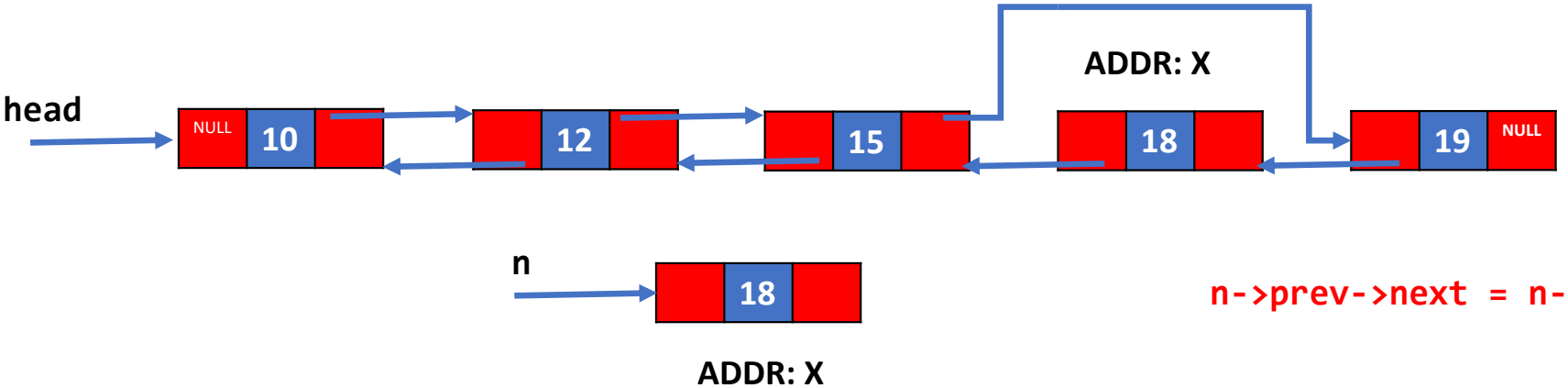
- After deletion



Delete a given node at pos > 0

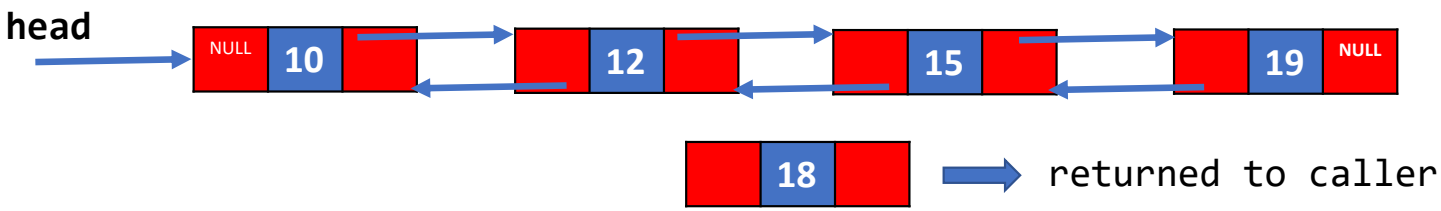
```
struct node {
   int val;
   struct node *next;
   struct node *prev;
};
```

- Before deletion




```
n->prev->next = n->next;
```

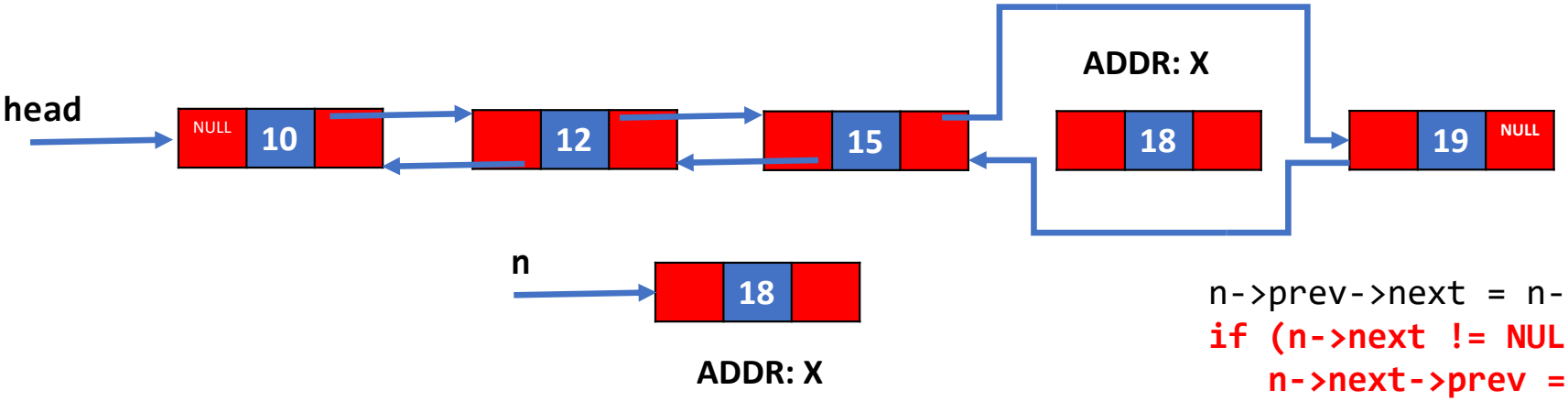
- After deletion



Delete a given node at pos > 0

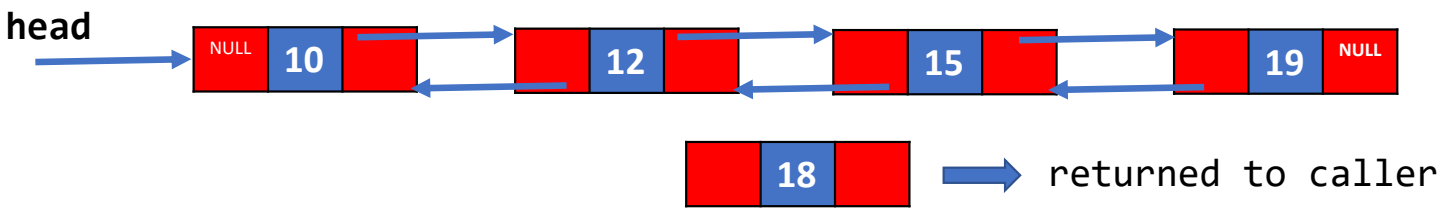
```
struct node {
   int val;
   struct node *next;
   struct node *prev;
};
```

- Before deletion




```
n->prev->next = n->next;
if (n->next != NULL) {
   n->next->prev = n->prev;
}
```

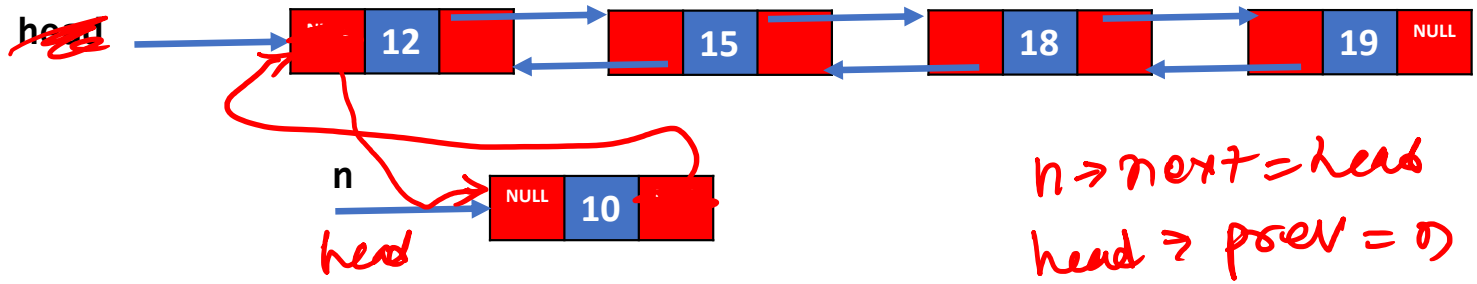
- After deletion



Insert (front)

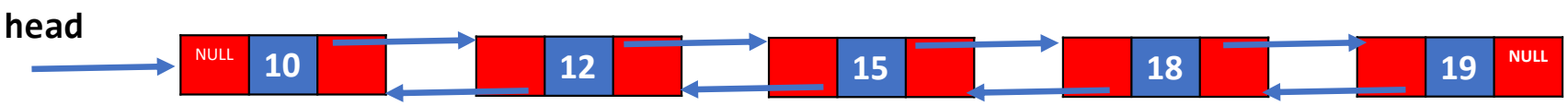
```
struct node {
    int val;
    struct node *next;
    struct node *prev;
};
```

- Before insertion



n->next=head
head->prev=n
head=n

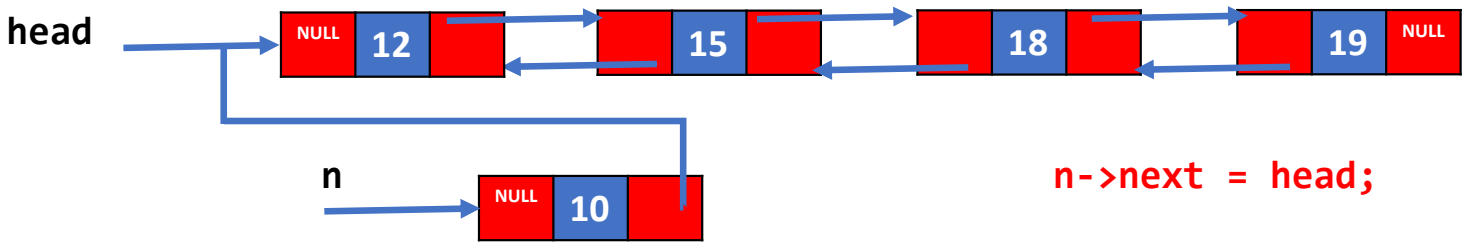
- After insertion



Insert (front)

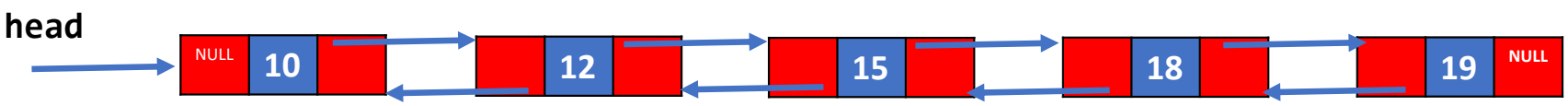
```
struct node {
   int val;
   struct node *next;
   struct node *prev;
};
```

- Before insertion




```
n->next = head;
```

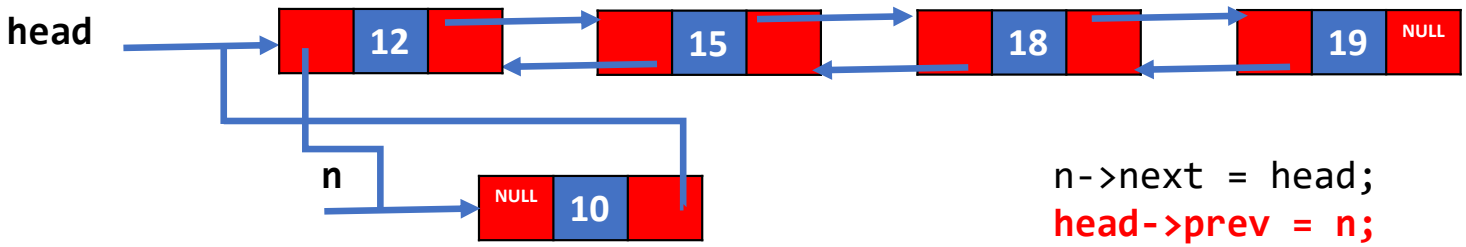
- After insertion



Insert (front)

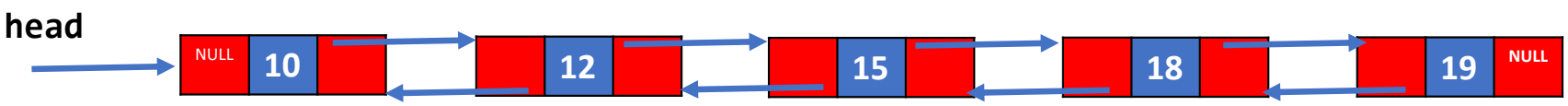
```
struct node {
    int val;
    struct node *next;
    struct node *prev;
};
```

- Before insertion




```
n->next = head;
head->prev = n;
```

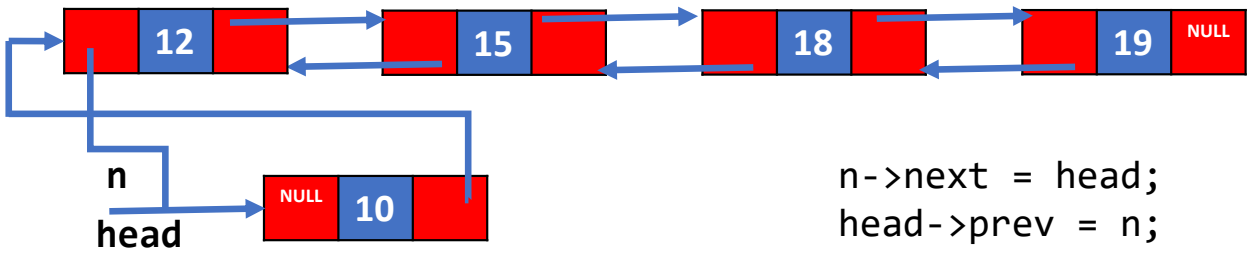
- After insertion



Insert (front)

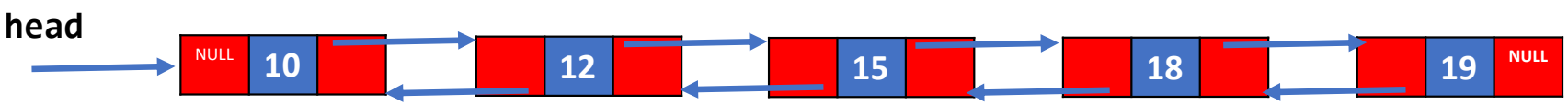
```
struct node {
    int val;
    struct node *next;
    struct node *prev;
};
```

- Before insertion




```
n->next = head;
head->prev = n;
head = n;
```

- After insertion



Doubly linked list

- You can try to implement other operations yourself that we did for the singly linked list

Doubly linked list

- In a doubly linked list, head points to the first node of the linked list (similar to a singly linked list)

- At any node at poison `i`, we can traverse in both directions, i.e., the node at positions `i-1` and `i+1` can be reached in one step

- This data structure efficiently handles the case when given a node we want to access nearby nodes in both directions or delete the node
  - e.g., Redo/Undo operations, Playlist with next and previous buttons

Doubly linked list

- Every node in a doubly linked list contains two references, `prev` and `next`, corresponding to the previous and next node

- The `prev` and `next` pointers of the first and last nodes point to the empty list (NULL)

Today’s topics

- Stacks
- Queues
- Trees

Announcement

- Makeup class on
  - **Wednesday (8 June)** from **4 pm - 5:30 pm** in **C102**.

Applications of stack

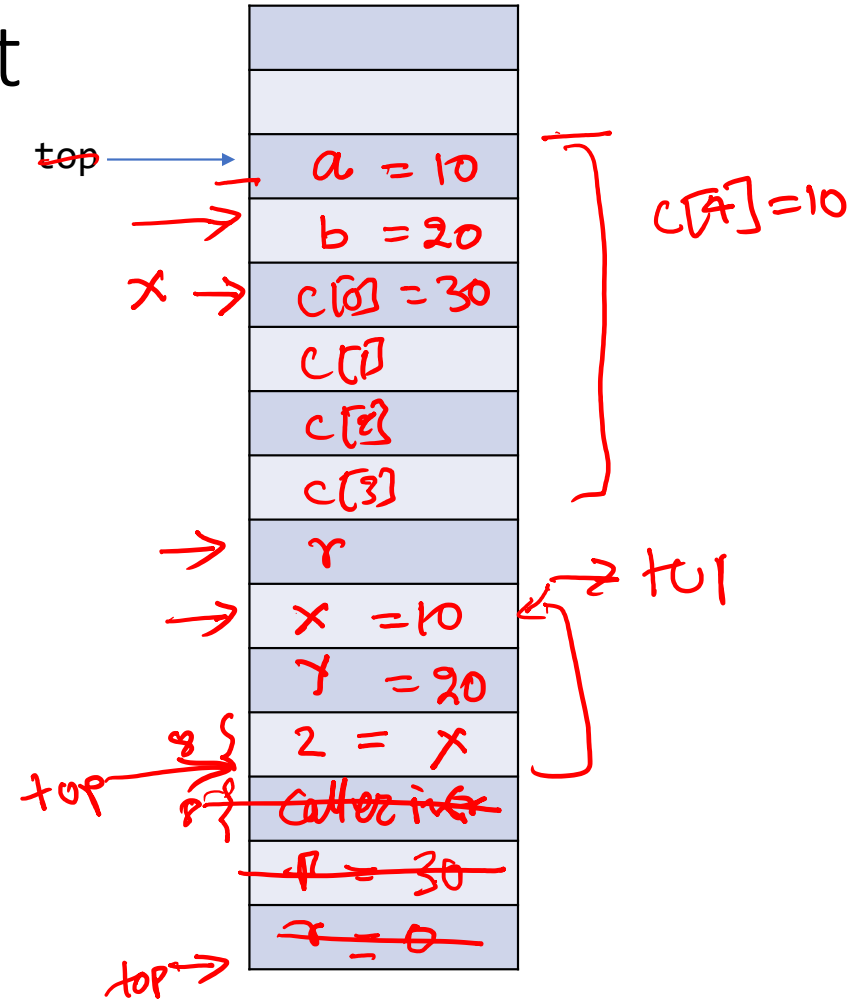
Local variables management

```
int foo(int x, int y, int *z) {
    int p = x + y;
    int r = p - z[0];
    return r;
}
```

```
int main() {
    int a = 10;
    int b = 20;
    int c[4], r;
    c[0] = a + b;
    r = foo(a, b, c);
    return r;
}
```

top points to first available index.

Let's say each slot is four bytes long.



Local variables management

- Have you ever noticed that the value of some other variable is changed when you do some out-of-bound array access?

Local variables management

- Have you ever encountered a stack overflow error while running your program?

Local variables management

- Have you ever encountered a stack overflow error while running your program?
  - The application stack is implemented using a fixed-size array
  - The array size is big enough to support most applications
  - You can change the default size of the stack using some compiler option

Matching symbols

- Three kinds of symbols
  - Parentheses: “(” and “)”
  - Braces: “{” and “}”
  - Brackets: “[” and “]”

- Goal: check if each opening symbol matches the corresponding closing symbol

Matching symbols

- Correct: (a + b) + (e * (d - {c + {a / b}})) + [c * d] {x + [y - z]}
- Incorrect:) (a + b)(
- Incorrect: {a + (b + c})

Basic idea

- Scan the string from left to right
- If an opening symbol is encountered, push it to stack
- If a closing symbol is encountered
  - If the stack is empty, return false
  - pop a symbol from the stack
  - if the popped symbol is not equal to the corresponding closing symbol, return false
- Skip other characters
- If the end of input is reached
  - If the stack is not empty, return false; otherwise, return true

Matching symbols

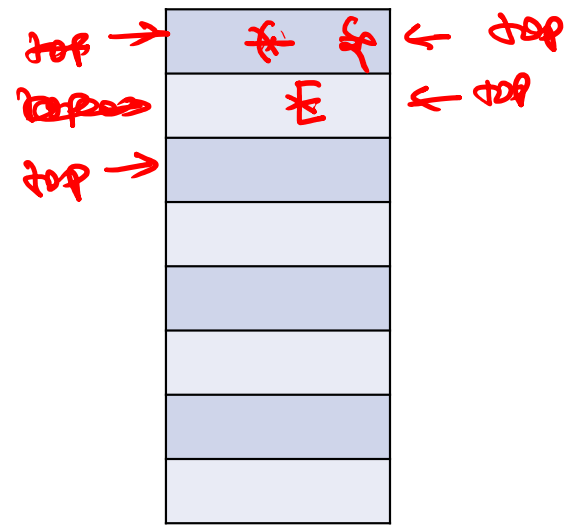
(a + b) * {[c + d]} * z

)

pop() = '('

pop() = '['

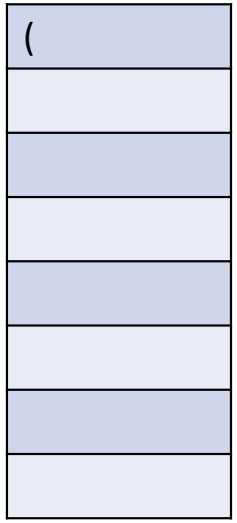
pop() = '{'



Matching symbols

(a + b) * {[c + d]} * z

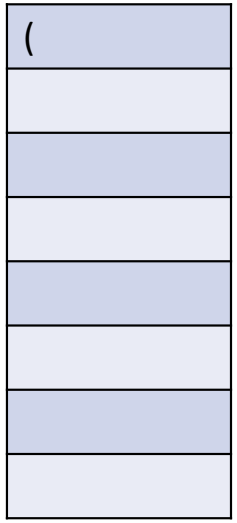
push (



Matching symbols

(a + b) * {[c + d]} * z

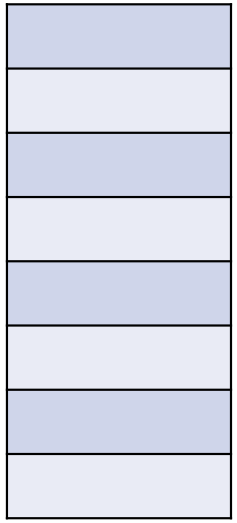
push (
skip a
skip +
skip b



Matching symbols

(a + b) * {[c + d]} * z

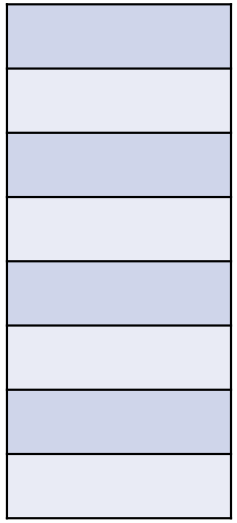
push (
skip a
skip +
skip b
pop == (



Matching symbols

(a + b) * {[c + d]} * z

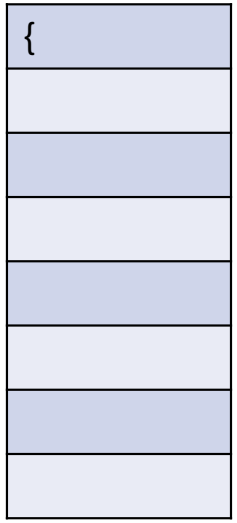
push (
skip a
skip +
skip b
pop == (
skip *



Matching symbols

(a + b) * {[c + d]} * z

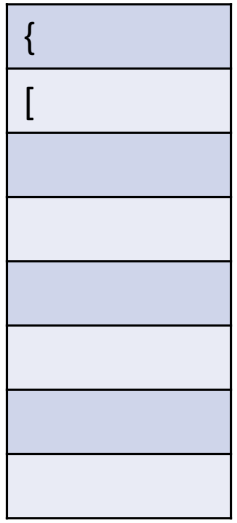
push (
skip a
skip +
skip b
pop == (
skip *
push {



Matching symbols

(a + b) * {[c + d]} * z

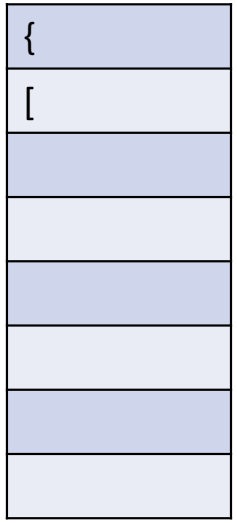
push (
skip a
skip +
skip b
pop == (
skip *
push {
push [



Matching symbols

(a + b) * {[c + d]} * z

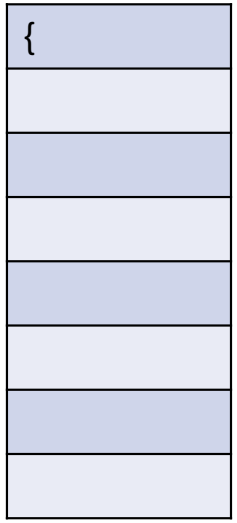
push (
skip a
skip +
skip b
pop == (
skip *
push {
push [
skip c
skip +
skip d



Matching symbols

(a + b) * {[c + d]} * z

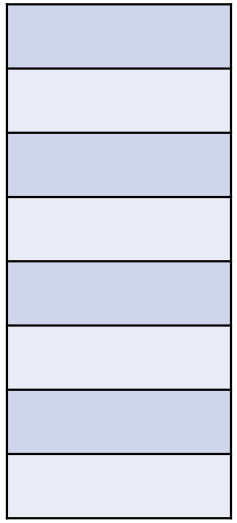
push (
skip a
skip +
skip b
pop == (
skip *
push {
push [
skip c
skip +
skip d
pop == [



Matching symbols

(a + b) * {[c + d]} * z

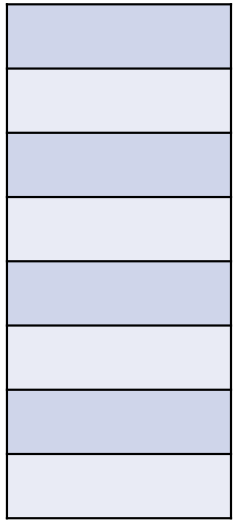
push (
skip a
skip +
skip b
pop == (
skip *
push {
push [
skip c
skip +
skip d
pop == [
pop == {



Matching symbols

(a + b) * {[c + d]} * z

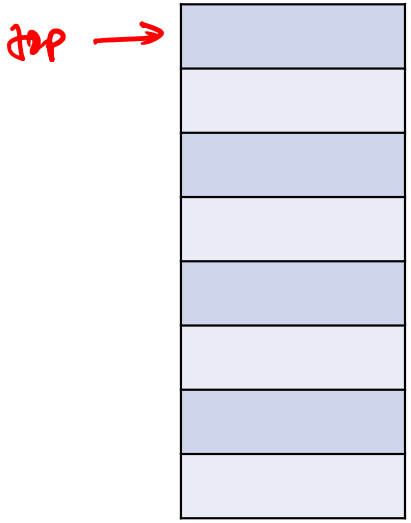
```
push (
skip a
skip +
skip b
pop == (
skip *
push {
push [
skip c
skip +
skip d
pop == [
pop == {
skip *
skip z
stack_empty == TRUE
```



Matching symbols

)(a + b)

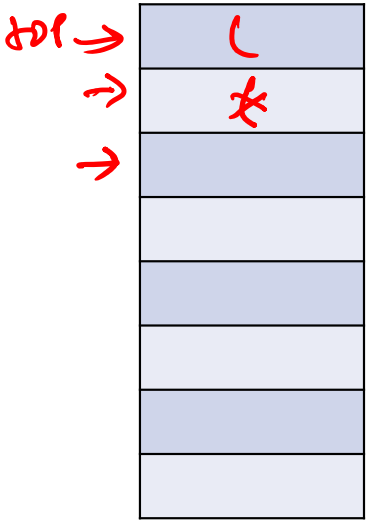
)



Matching symbols

((a + b)

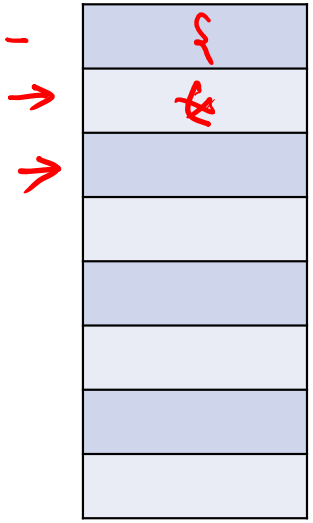
pop() = (



Matching symbols

{(a + b})

pop == '{'



Pseudo-code

- Read section-1.7.2 from Goodrich et al. book

Pseudo-code

- Pseudo-code is a mixture of programming language constructs and natural language
  - In pseudo-code, we can mix C semantics with a little bit of natural language for brevity
- There is no precise definition of a pseudo-code because of its dependence on the natural language
- Next, we are going to define the syntax of pseudo-code that we will follow in this course

Pseudo-code

- Assignment : ← instead of =
- Comparison : = instead of ==
- Function declaration: **Algorithm** name(arg1, arg2, …)
  - followed by the description of input parameters and the output
- Conditional : **if** condition **then** true-actions **else** false-actions
  - Use indentation instead of braces
- While loops: **while** condition **do** actions
- For loops: **for** variable-increment-definition **do** actions
- Array indexing: **A[i]** // same as C
- Function calls: **method**(arg1, arg2) // same as C
- Function return: **return** value // same as C
- Structure accesses: n->next, var.x, etc. // same as C

Matching symbols

typeof (char c)
{ if (c == '(' || c == ')')
return 1;
else if (c == '[' || c == ']')
return 2.

```
Algorithm MachingSymbols(Str, n):
  Input: An array Str of n characters
  Output: true only if all symbols are matching

  Let typeof('(') = typeof(')') = 1
  Let typeof('[') = typeof(']') = 2
  Let typeof('{') = typeof('}') = 3
  Let S is an empty stack
  for i ← 0 to n -1 do
    if Str[i] belongs to set ('(', '{', '[') then
      S.push(typeof(Str[i]))
    else if Str[i] belongs to set (')', '}', ']') then
      if S.isEmpty() then
        return false
      else if S.pop() != typeof(Str[i]) then
        return false
  if S.isEmpty() then
    return true
  else
    return false
```

Queues

References

- Read section-10.1 from Cormem et al.
- Read section-5.2 from Goodrich et al.

Queue example

This Photo by Unknown Author is licensed under CC BY

Queue

- Queue is the collection of items
- Queue has two ends head and tail
- Insert items at the head of the queue
- Remove items from the tail of the queue
- Implements first-in, first-out strategy

Queue

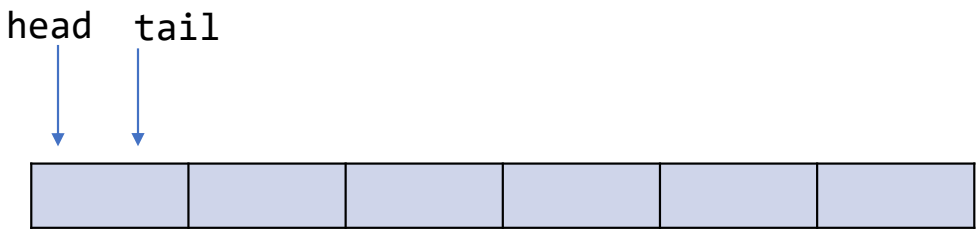
- Follow first-in, first-out (FIFO) policy
- Queue ADT:
  - QUEUE-EMPTY => returns true if the queue is empty
  - ENQUEUE => insert an item at the tail of the queue
  - DEQUEUE => remove and return the item at the head of the queue
  - FIRST => return the first item in the queue without removing

Example

Operations	Output	Queue contents
enqueue(5)	–	(5)
enqueue(3)	–	(5 3)
dequeue()	5	(3)
enqueue(7)	–	(3 7)
dequeue()	3	(7)
first()	7	(7)
dequeue()	7	()
dequeue()	underflow	()
first()	underflow	()
queue_empty()	true	()
enqueue(9)	–	(9)
enqueue(7)	–	(9 7)
enqueue(7)	–	(9 7 7)
dequeue()	9	(7 7)
queue_empty()	false	(7,7)

Queue using array

Queue

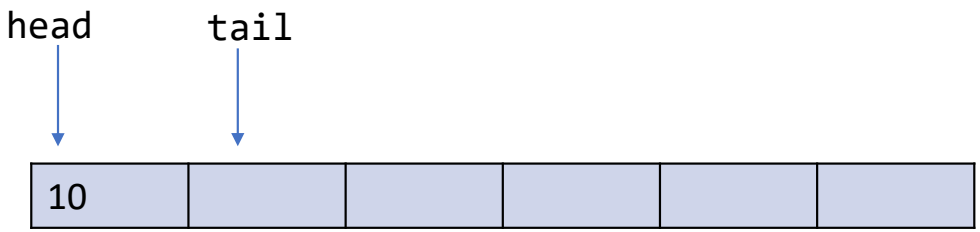


queue_empty:

head == tail

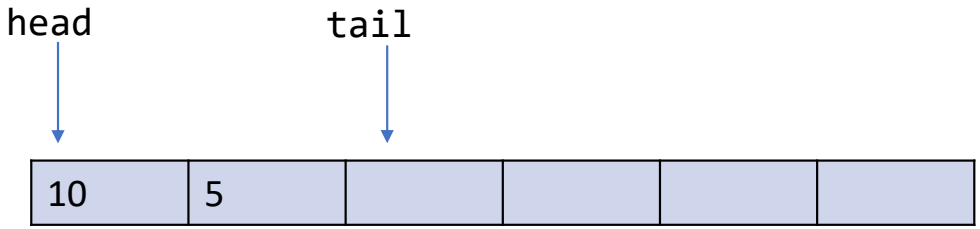
```
Queue is empty.
Queue size = 6
```

Queue



enqueue(10)

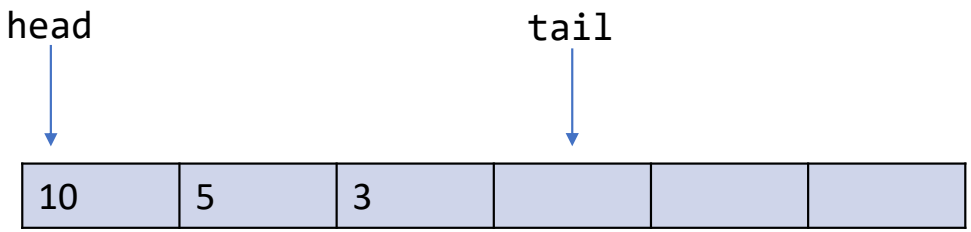
Queue




```
enqueue(10)
enqueue(5)
```

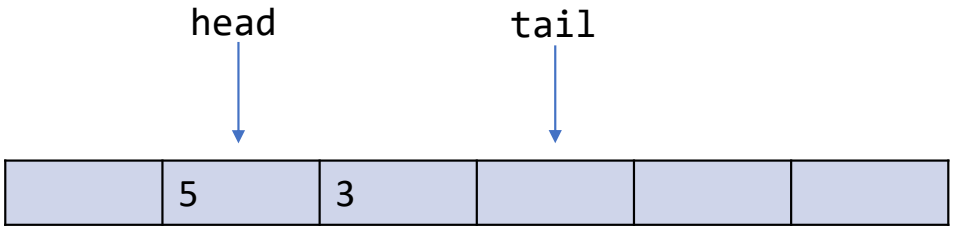
```
next_tail(tail):
    return tail+1

enqueue: (arr, tail, val)
    arr[tail] = val
    tail = next_tail(tail)
```

Queue



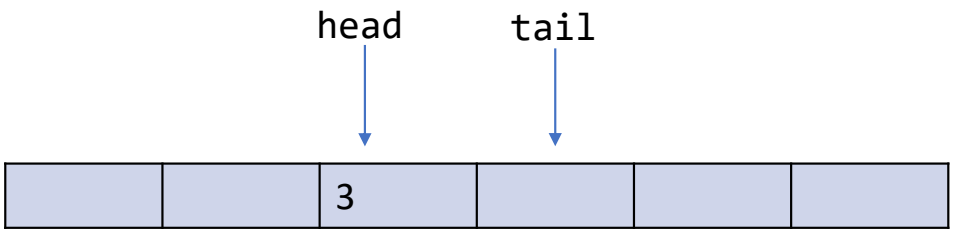

```
enqueue(10)
enqueue(5)
enqueue(3)
```

Queue




```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
```

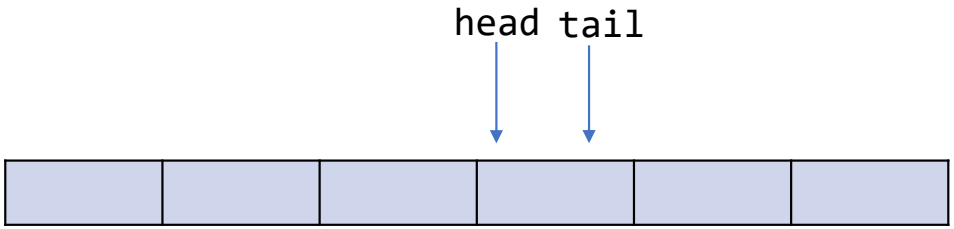
Queue




```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
```

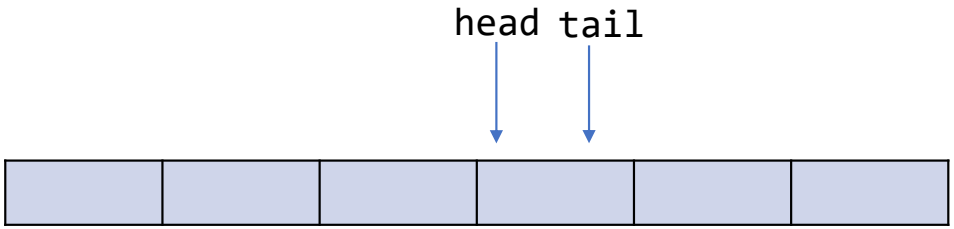
```
next_head(head):
    return head+1

dequeue: (arr, head)
    val = arr[head]
    head = next_head(head)
    return val
```

Queue




```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
```

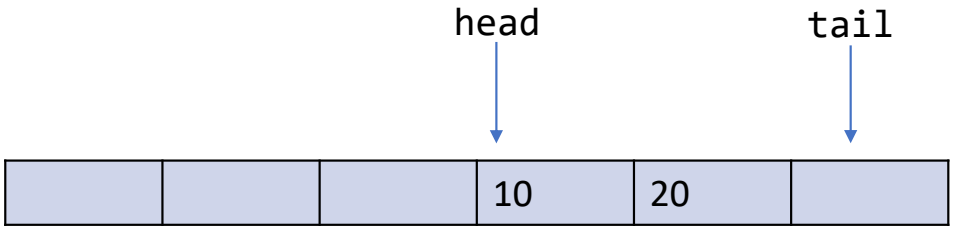
Queue




```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
next_head(head):

dequeue:
    if (queue_is_empty())
        Print (underflow);
```

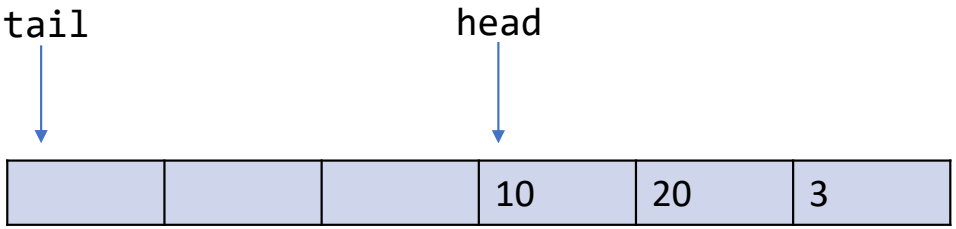
Queue




```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = “Underflow”
```

```
enqueue(10)
enqueue(20)
```

Queue

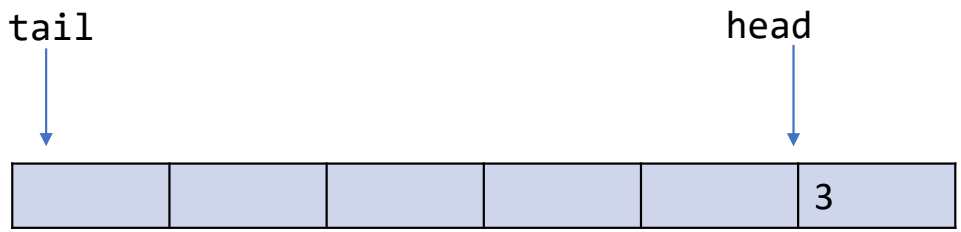


next_tail(tail): size
if (tail == size-1)
return 0;
else return tail+1;
enqueue:

```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
```

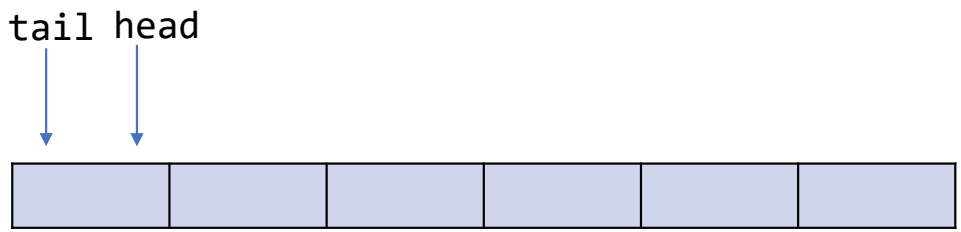
Queue




```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
dequeue() = 10
dequeue() = 20
```

Queue



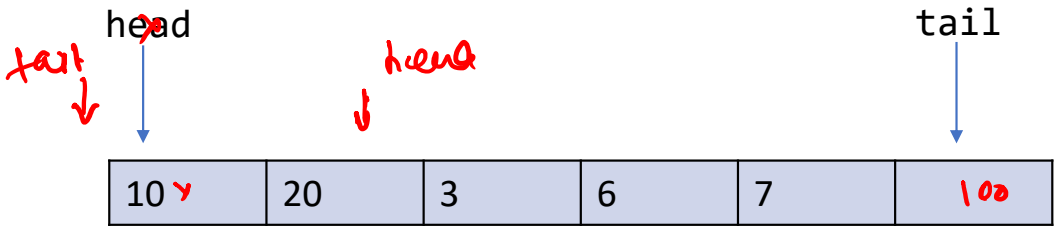
next_head(~~tail~~ head, size):
    if (head == size-1)
        return 0;
    else return head+1;

dequeue:

```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
dequeue() = 10
dequeue() = 20
dequeue() = 3
```

Queue

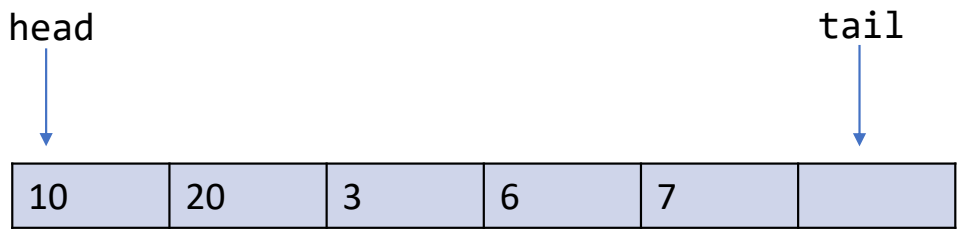



```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
dequeue() = 10
dequeue() = 20
dequeue() = 3
```

```
enqueue(10)
enqueue(20)
enqueue(3)
enqueue(6)
enqueue(7)
```

Queue



queue_overflow:
if (next_tail(tail, size) == head)
return true;
else return false;

```
enqueue(10)
enqueue(5)
enqueue(3)
dequeue() = 10
dequeue() = 5
dequeue() = 3
dequeue() = "Underflow"
```

```
enqueue(10)
enqueue(20)
enqueue(3)
dequeue() = 10
dequeue() = 20
dequeue() = 3
```

```
enqueue(10)
enqueue(20)
enqueue(3)
enqueue(6)
enqueue(7)
enqueue(9) "overflow"
```

Type of queue

```
struct queue {
    int tail;
    int head;
    int size;

    int *arr;
};
```

Type of queue

```
struct queue {
  int head;
  int tail;
  int size;
  int *arr;
};
```

Initialize queue

```
struct queue* initialize_queue(int size) {
  struct queue *Q = (struct queue*)malloc(sizeof(struct queue));
  assert(Q);
  Q->arr = (int*)malloc(size * sizeof(int));
  assert(Q->arr);
  Q->head = Q->tail = 0;
  Q->size = size;
  return Q;
}
```

Queue-empty

```
int queue_empty(struct queue *Q) {
  if (Q->head == Q->tail) {
    return 1;
  }
  return 0;
}
```

Next-head

```
int next_head(struct queue *Q)
{
  if (Q->head == Q->size - 1) {
    return 0;
  }
  return Q->head + 1;
}
```

Next-tail

```
int next_tail(struct queue *Q)
{
  if (Q->tail == Q->size - 1) {
    return 0;
  }
  return Q->tail + 1;
}
```

Queue-full

```
int queue_full(struct queue *Q) {
  if (next_tail(Q) == Q->head) {
    return 1;
  }
  return 0;
}
```

Enqueue

```
void enqueue(struct queue *Q, int val) {
  if (queue_full(Q)) {
    printf("Queue overflow\n");
    exit(0);
  }
  Q->arr[Q->tail] = val;
  Q->tail = next_tail(Q);
}
```

Dequeue

```
int dequeue(struct queue *Q) {
  if (queue_empty(Q)) {
    printf("Queue underflow\n");
    exit(0);
  }
  int ret = Q->arr[Q->head];
  Q->head = next_head(Q);
  return ret;
}
```

Example

```
struct queue {
  int head;
  int tail;
  int size;
  int *arr;
};
```

```
int main() {
  int i;
  struct queue *Q = initialize_queue(10);

  assert(queue_empty(Q));
  for (i = 0; i < 9; i++) {
    enqueue(Q, i);
    print_queue(Q);
  }
  assert(!queue_empty(Q));
  for (i = 0; i < 9; i++) {
    dequeue(Q);
    print_queue(Q);
  }
  assert(queue_empty(Q));
  uninitialize_queue(Q);
  return 0;
}
```

Exercise

- Implement an API to print all elements in a queue starting from head to tail

- Implement `FIRST(Q)` API from the queue ADT

Queue using linked list

Queue

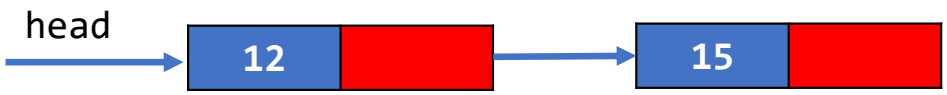
head = NULL

```
Empty queue
```

Enqueue

```
enqueue(12)
```

Enqueue



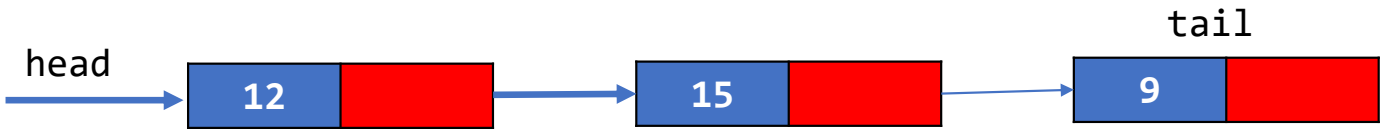

```
enqueue(12)
enqueue(15)
```

Enqueue

```
enqueue(12)
enqueue(15)
enqueue(9)
```

Which list operation is suitable for enqueue?
insert_front()
insert_rear()

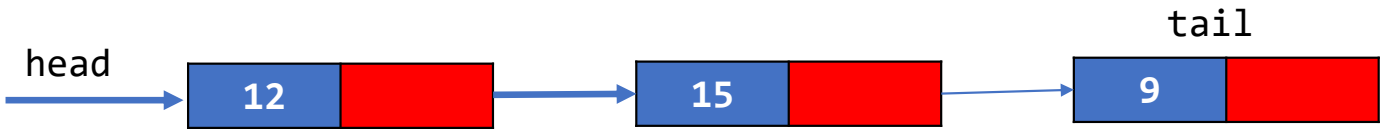
Enqueue



```
enqueue(12)
enqueue(15)
enqueue(9)
enqueue(5)
```

tail->next = n
tail = n

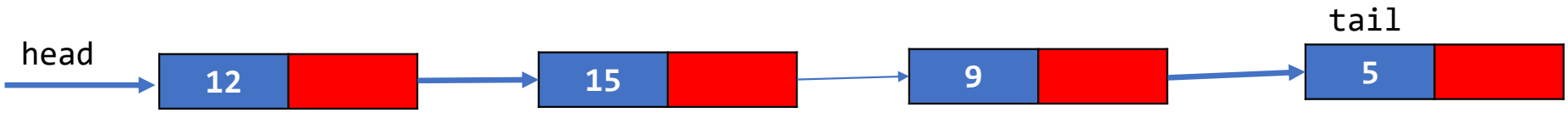
Enqueue



```
enqueue(12)
enqueue(15)
enqueue(9)
enqueue(5)
```

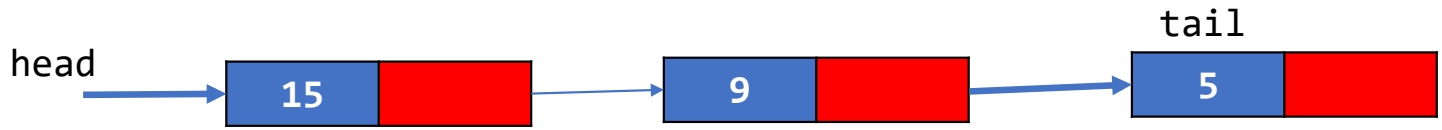
```
tail->next = n
tail = n
```

Enqueue



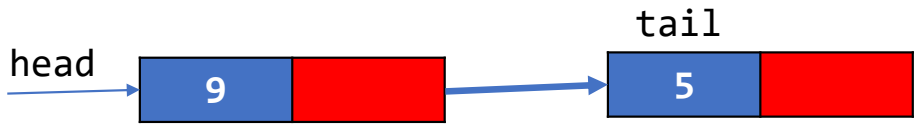

```
enqueue(12)
enqueue(15)
enqueue(9)
enqueue(5)
```

Dequeue




```
enqueue(12)
enqueue(15)
enqueue(9)
enqueue(5)
dequeue() = 12
```

Dequeue




```
ret = delete_front(&head)
```

```
enqueue(12)
enqueue(15)
enqueue(9)
enqueue(5)
dequeue() = 12
dequeue() = 15
```

Dequeue

```
enqueue(12)
enqueue(15)
enqueue(9)
enqueue(5)
dequeue() = 12
dequeue() = 15
dequeue() = 9
dequeue() = 5
```

Dequeue

```
enqueue(12)
enqueue(15)
enqueue(9)
enqueue(5)
dequeue() = 12
dequeue() = 15
dequeue() = 9
dequeue() = 5
dequeue() = “underflow”
```

Type

```
struct queue {
    struct node *head;
    struct node *tail;

};
```

Type

```
struct queue {
  struct node *head;
  struct node *tail;
};
```

Initialize queue

```
struct queue* initialize_queue() {
  struct queue *Q = (struct queue*)malloc(sizeof(struct queue));
  assert(Q);
  Q->head = NULL;
  Q->tail = NULL;
  return Q;
}
```

Queue-empty

```
int queue_empty(struct queue *Q) {
  if (Q->head == NULL) {
    return 1;
  }
  return 0;
}
```

Enqueue

```
void enqueue(struct queue *Q, int val) {
  struct node *n = allocate_node(val);
  if (queue_empty(Q)) {
    Q->head = Q->tail = n;
  }
  else {
    Q->tail->next = n;
    Q->tail = n;
  }
}
```

Dequeue

```
int dequeue(struct queue *Q) {
  if (queue_empty(Q)) {
    printf("Queue underflow\n");
    exit(0);
  }
  struct node *ret = delete_front(&Q->head);
  int retval = ret->val;
  free(ret);
  return retval;
}
```

Example

```
struct queue {
  struct node *head;
  struct node *tail;
};
```

```
int main() {
  int i;
  struct queue *Q = initialize_queue();

  assert(queue_empty(Q));
  for (i = 0; i < 9; i++) {
    enqueue(Q, i);
    print_queue(Q);
  }
  assert(!queue_empty(Q));
  for (i = 0; i < 9; i++) {
    dequeue(Q);
    print_queue(Q);
  }
  assert(queue_empty(Q));
  uninitialize_queue(Q);
  return 0;
}
```

Application of queues

- Train or flight booking software
- Buffer in devices, e.g., NIC, keyboard, etc.
- Scheduling of applications
- Access to shared resources
- Queue of network packets or disk blocks inside the OS

Today’s topics

- Trees
- Binary search trees

Tree

References

- Read section-4.3 from Udi Manber
- Read Chapter-12 from Cormen et al.
- Read Section-10.1 from Goodrich et al.

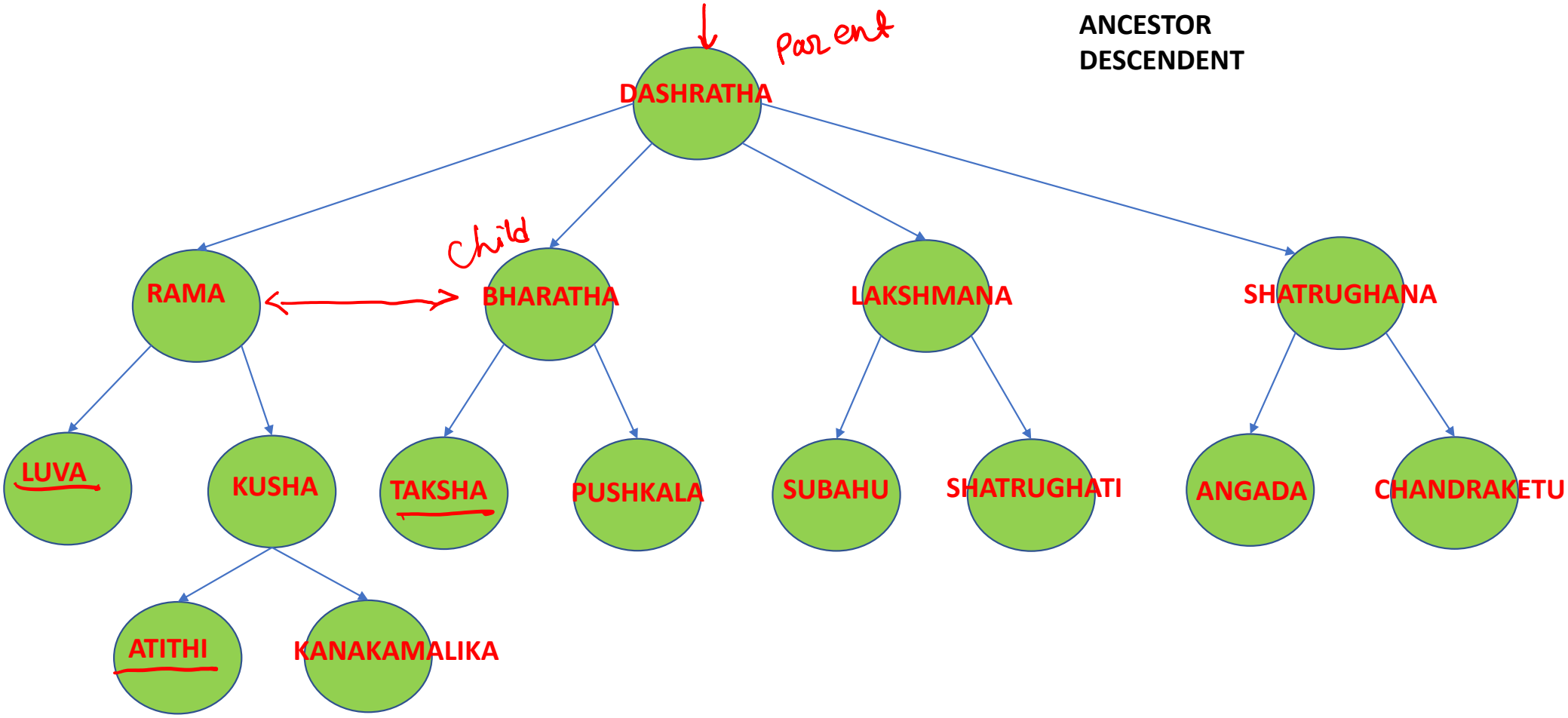
Linear data structures

- Array and Linked Lists are linear structures that contains a sequence of elements

- **Rooted tree** has non-linear (hierarchical) structure

Family tree (Ramayana)

PARENT
CHILDREN
SIBLING
ANCESTOR
DESCENDENT

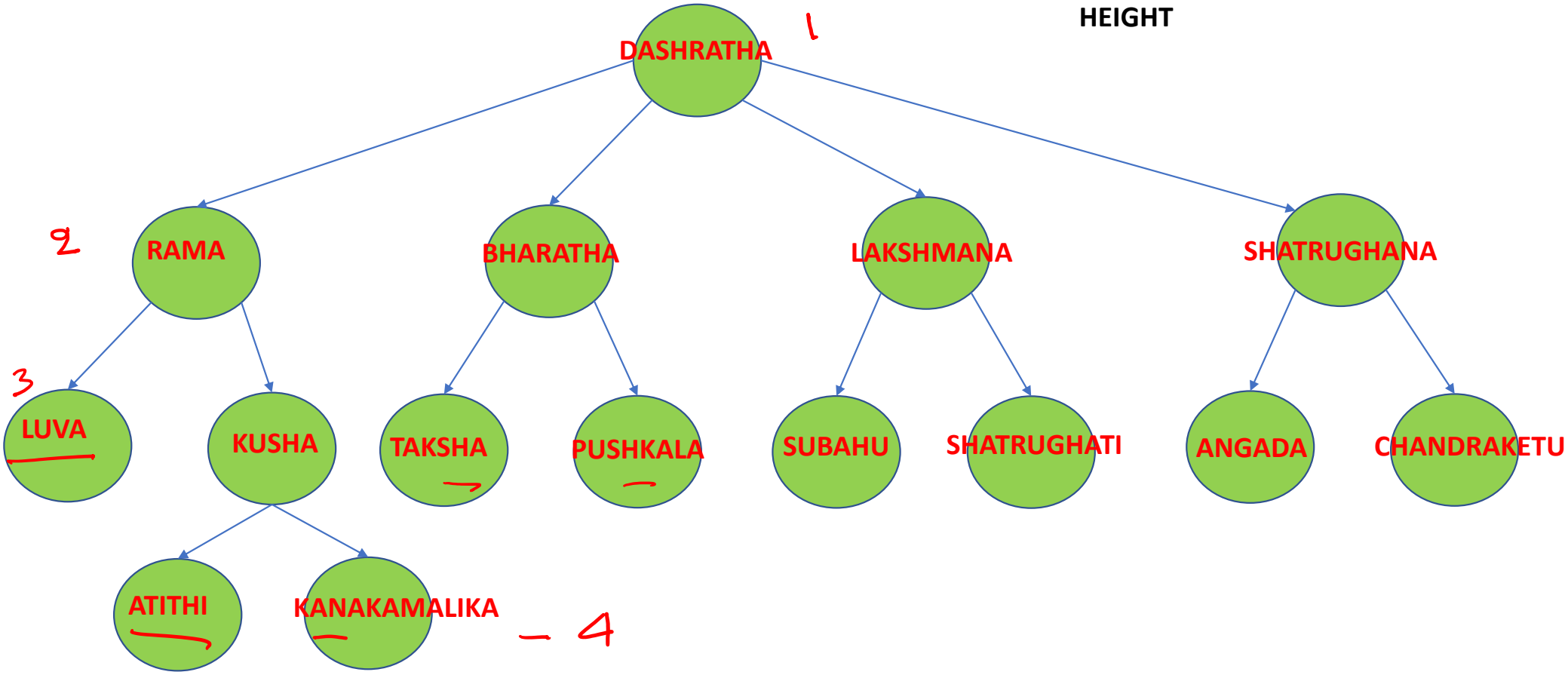


Tree structure

- The top element (node) of a non-empty tree is called the **root node**
- Each node (other than root) in the tree has exactly one **parent node**
- Each node in the tree has zero or more **children nodes**
- Trees are drawn upside down

Family tree (Ramayana)

LEAF
DEGREE
INTERNAL NODE
LEVEL
HEIGHT

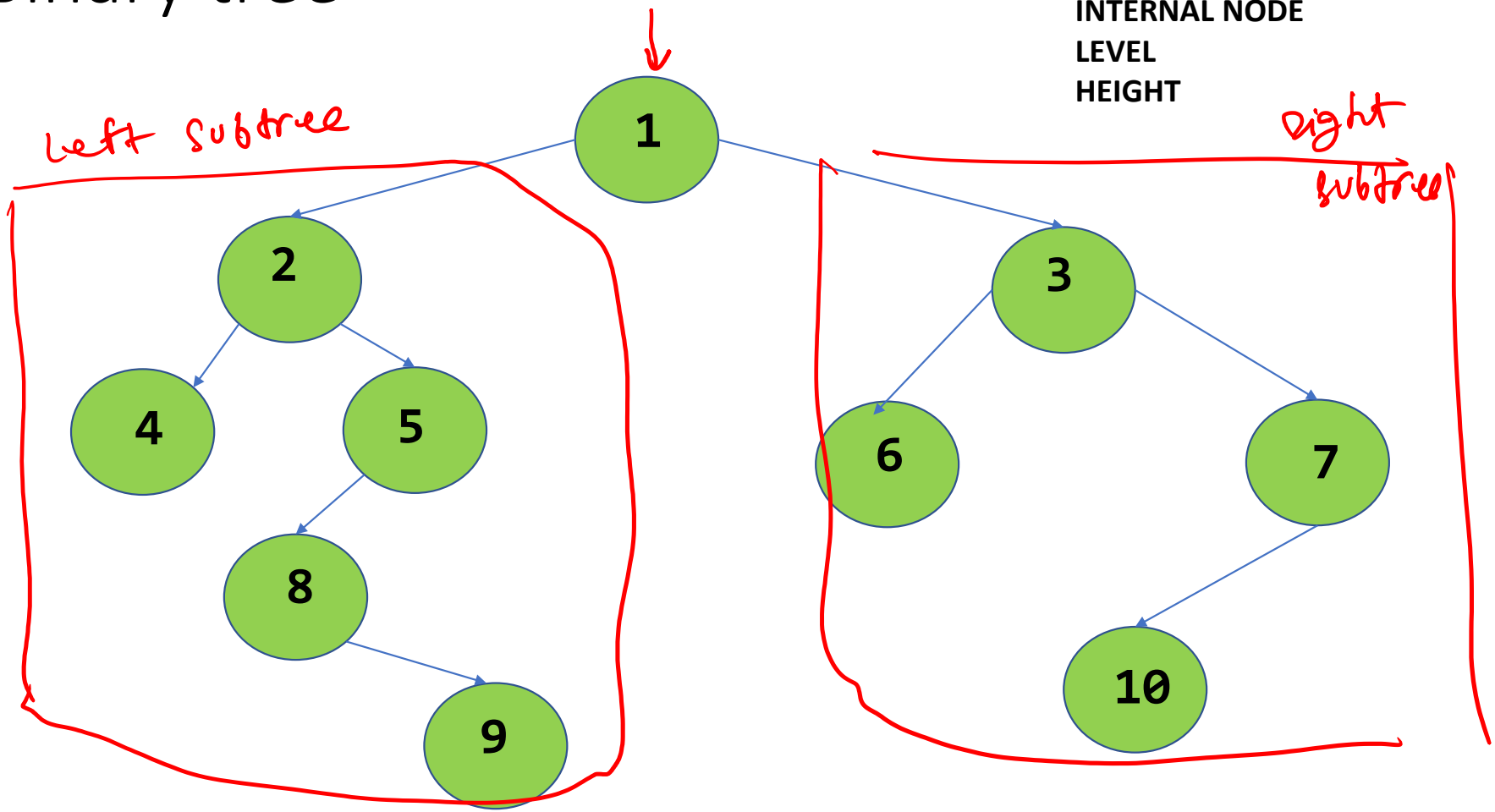


Terminology

- A node with no children is called a **leaf node**
- A node that is not a leaf is called an **internal node**
- The root node is present at **level-1**
- The root node is connected to nodes at level-2, level-2 nodes are connected to nodes at level-3, and so on
- The **height** of the tree is the maximum level of the tree
- The maximum children in any node is called the **degree** of a tree
- Trees of **degree two** are called **binary trees**

Binary tree

LEAF
DEGREE
INTERNAL NODE
LEVEL
HEIGHT



Binary tree

- A binary tree is either empty or it consist of
  - a root node
  - two distinct binary trees called the left and right subtrees
  - root contains references to left and right subtrees

Type (Binary Tree)

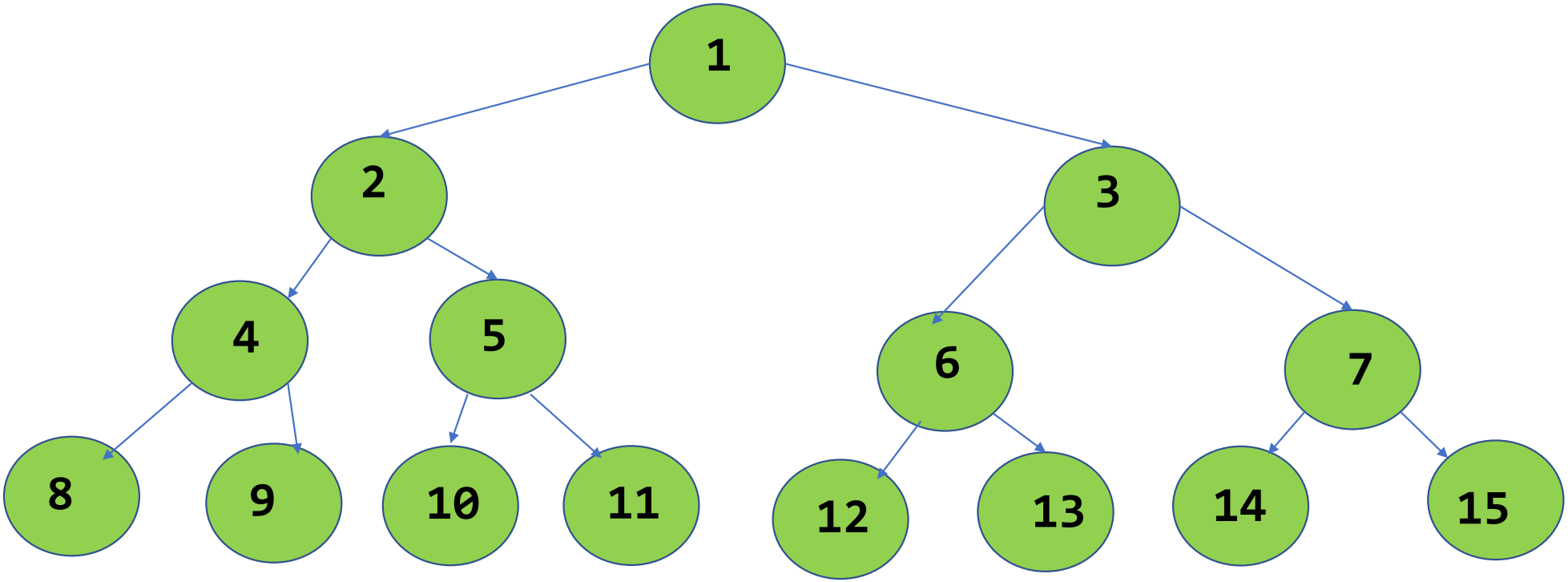
```
struct node {
    int val;
    struct node *left;
    struct node *right;

};
```

Type (General Tree)

```
struct node {
    int val;
    int degree;
    struct node **children;

};
```

Full binary tree

- In a full binary tree, each internal node has two children

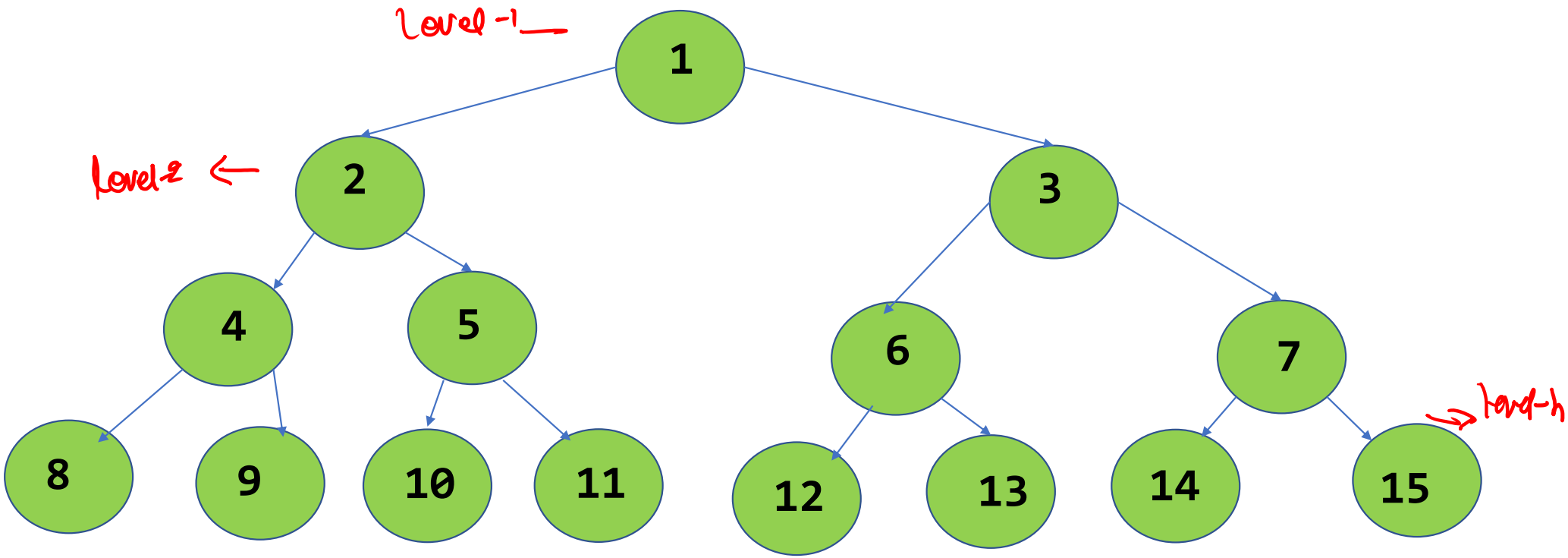


Height of a full binary tree

$2^0 + 2^1 + \dots + 2^{h-1} = n$

$2^h - 1 = n$

$h = \log_2(n+1)$



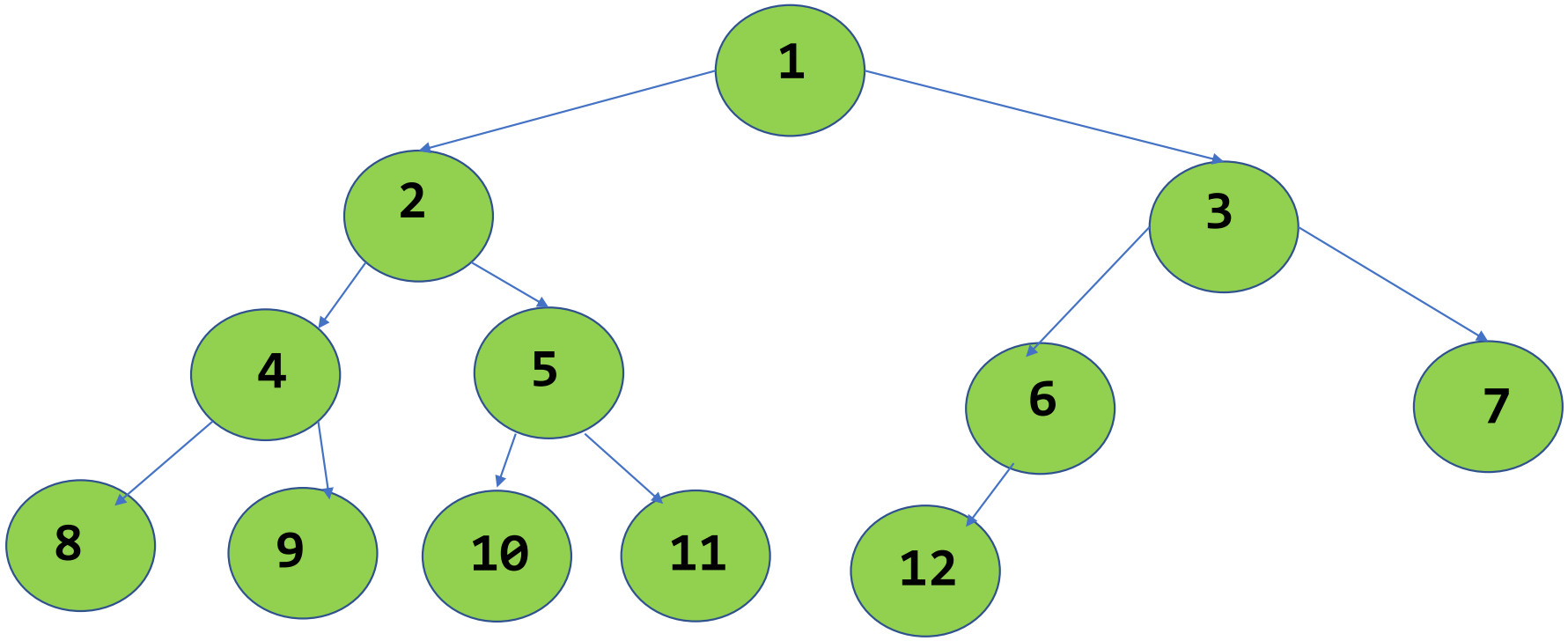
Height of a full binary tree

$$1 + 2 + 2^2 + \cdots + 2^{h-1} = n$$
$$2^h - 1 = n$$
$$h = \log_2(n + 1)$$

Complete binary tree

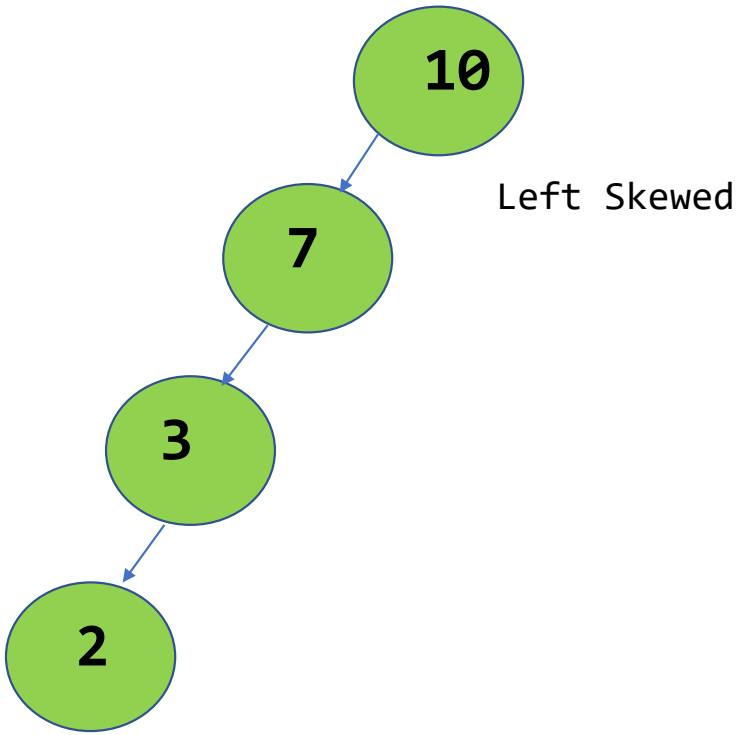
- In a complete binary tree, all the levels, except possibly the last level, are completely filled
- The nodes in the last level are filled from left the right
- If we remove all nodes from the last level, the resulting tree is a full binary tree

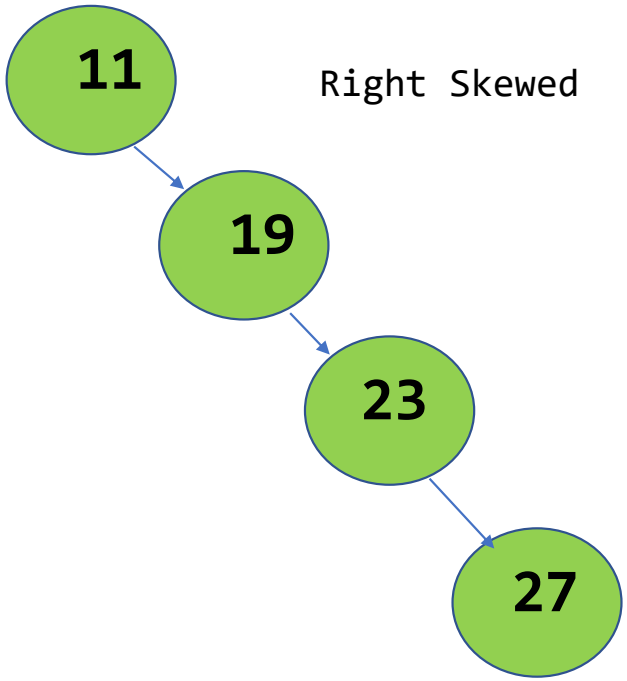
Complete binary tree



Skewed tree

- In a skewed binary tree, all the nodes have at most child





Tree traversal

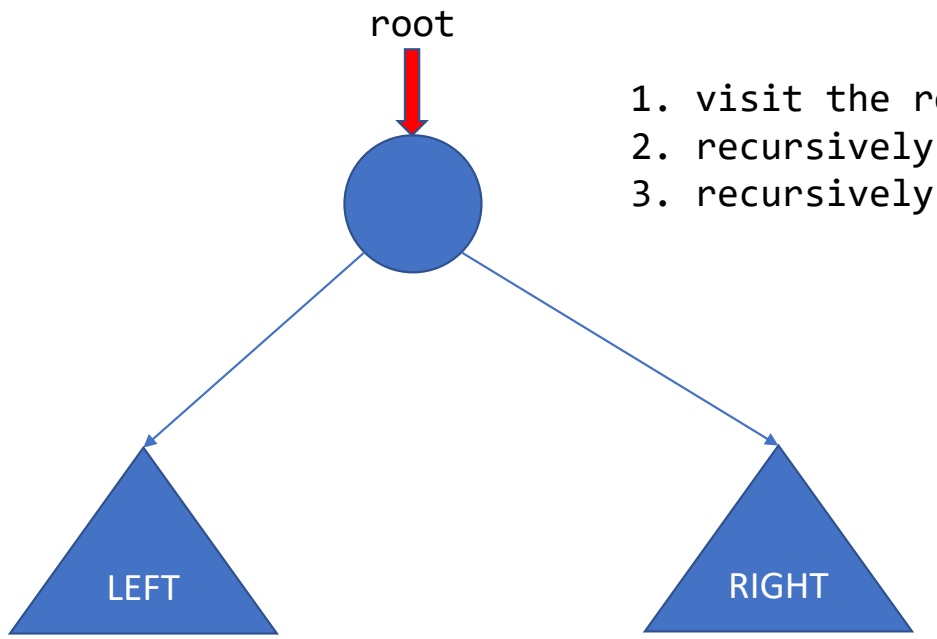
Tree traversal

- The aim of the traversal is to visit each node in the tree exactly once

- Generally, we traverse the tree in the following orders
  - Preorder traversal
  - Inorder traversal
  - Postorder traversal

- The above algorithms are recursive in nature

Preorder traversal

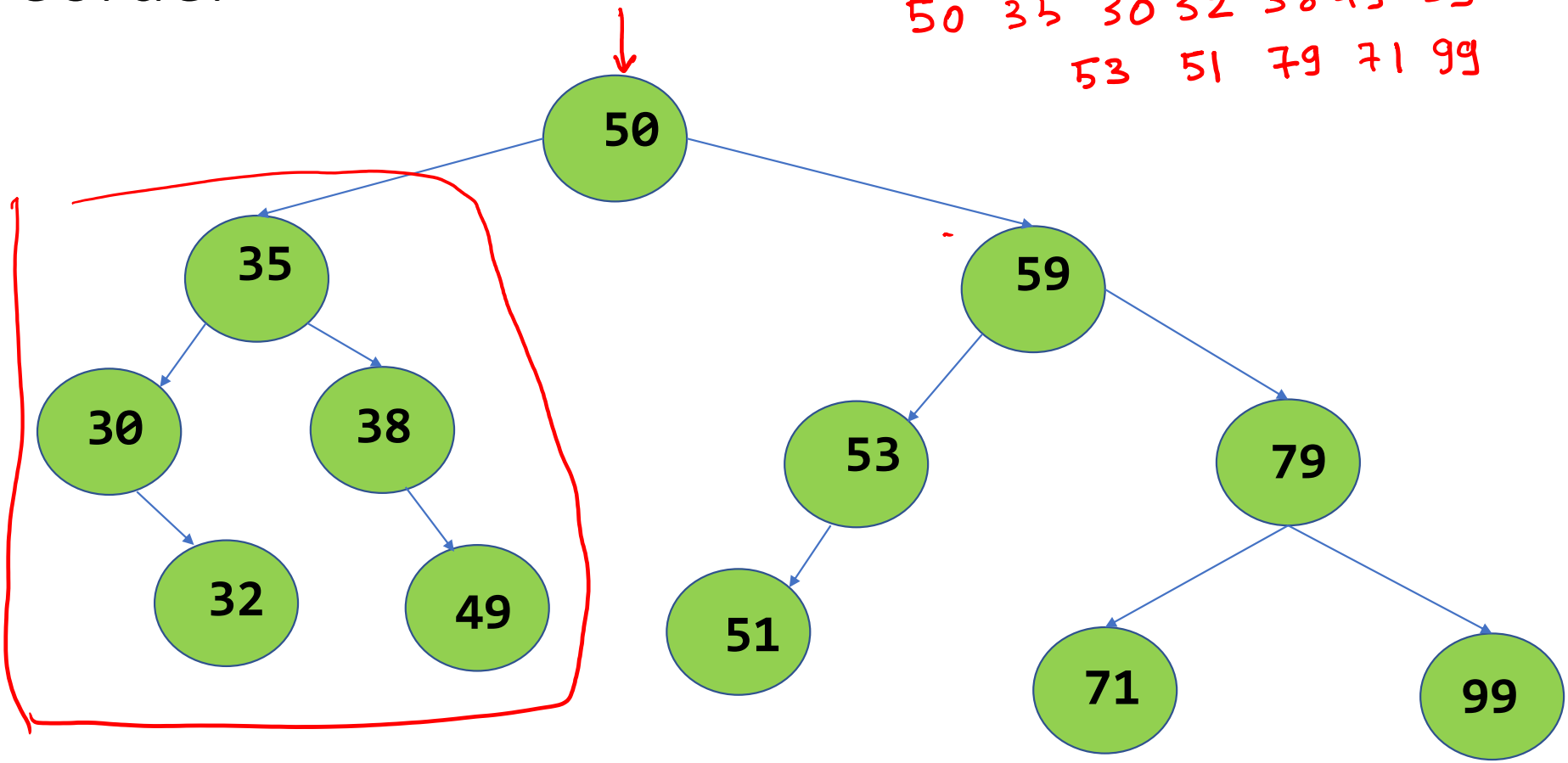


1. visit the root node
2. recursively traverse the left subtree
3. recursively traverse the right subtree

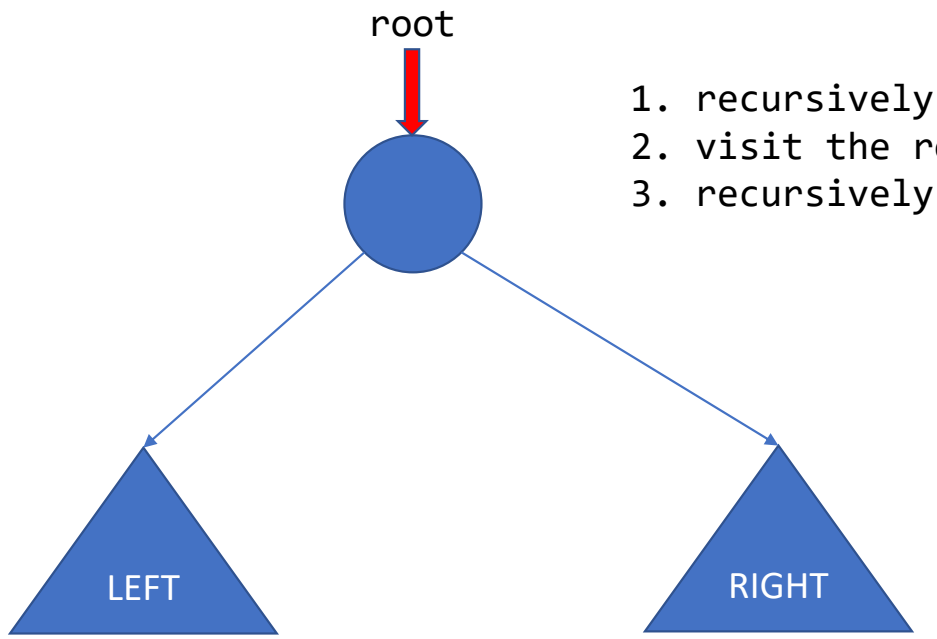
Preorder

PREORDER: ROOT LEFT RIGHT

50 35 30 32 38 49 59
53 51 79 71 99



Inorder traversal

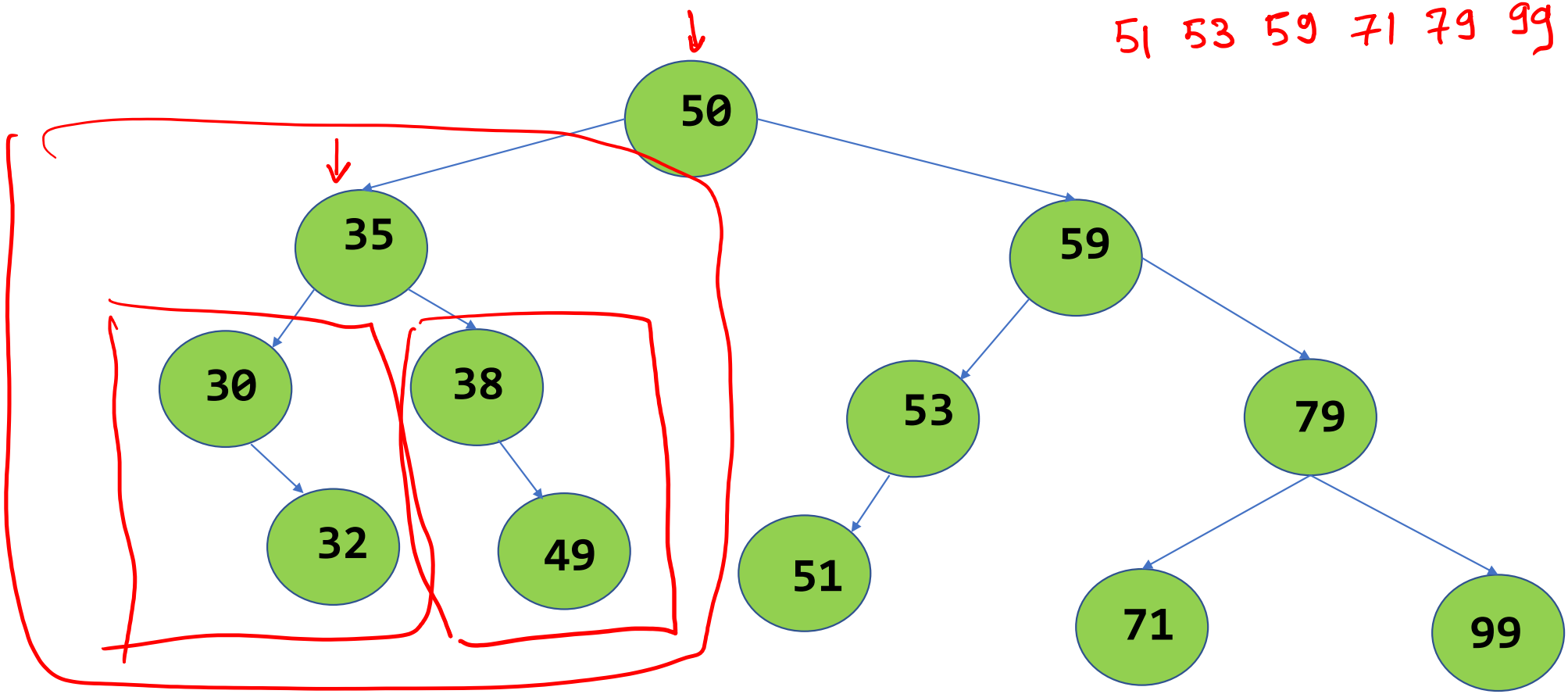



```
1. recursively traverse the left subtree
2. visit the root node
3. recursively traverse the right subtree
```

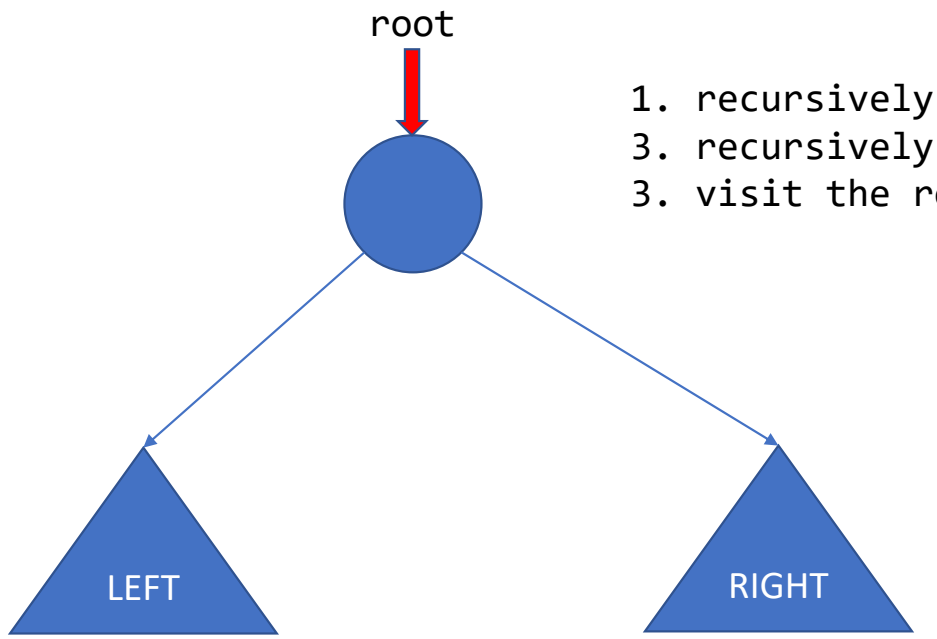
Inorder

INORDER: LEFT ROOT RIGHT

30 32 35 38 49 50
51 53 59 71 79 99



Postorder traversal

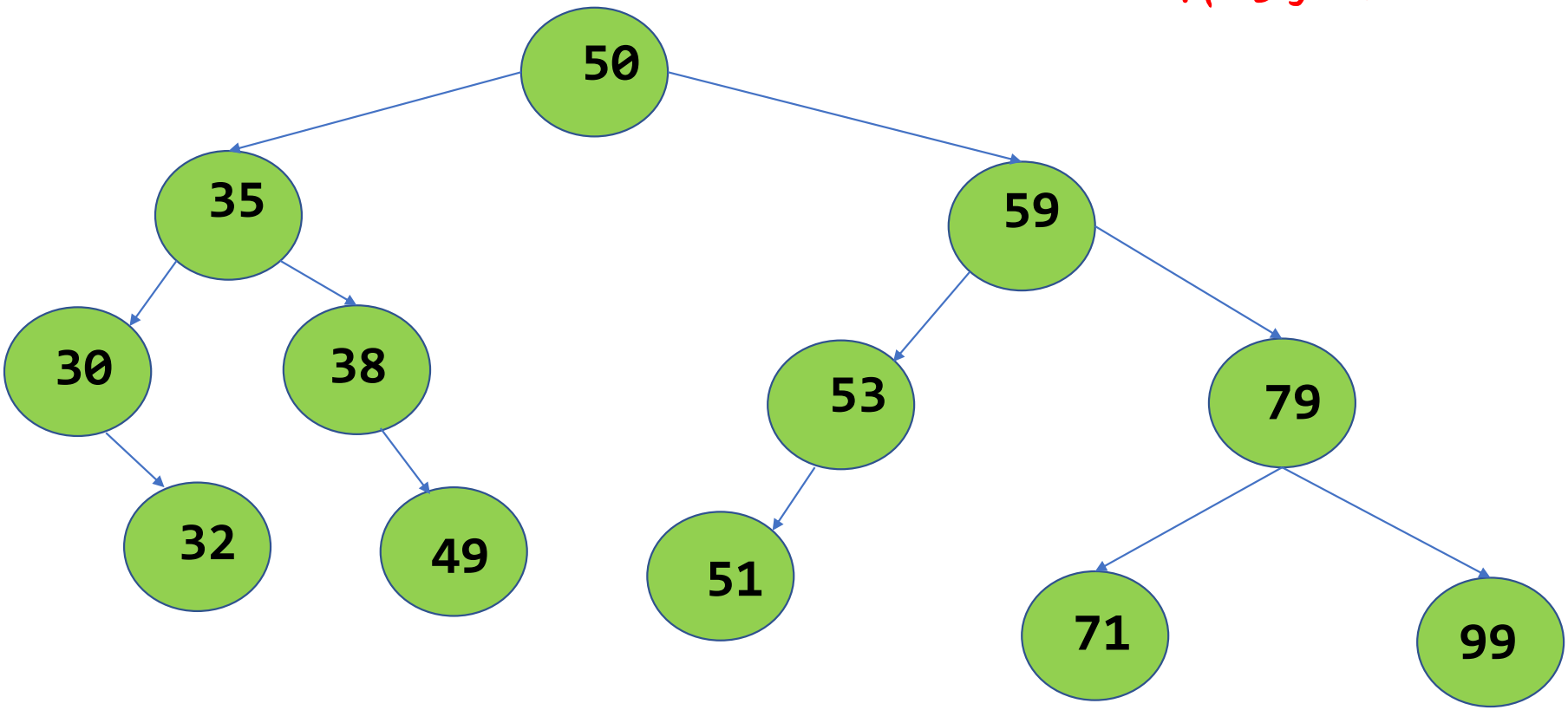



```
1. recursively traverse the left subtree
3. recursively traverse the right subtree
3. visit the root node
```

Postorder

POSTORDER: LEFT RIGHT ROOT

32 30 49 38 35 51 53
71 99 79 59 50



Preorder traversal

```
void preorder(struct node *root) {
  if (root == NULL) {
    return;
  }
  printf("%d ", root->val);
  preorder(root->left);
  preorder(root->right);
}
```

Time complexity:

$T(n) = T(n-1) + c$

$T(n) \leq 2T(n/2) + c$

$O(n)$

Inorder traversal

```
void inorder(struct node *root) {
  if (root == NULL) {
    return;
  }
  inorder(root->left);
  printf("%d ", root->val);
  inorder(root->right);
}
```

Postorder traversal

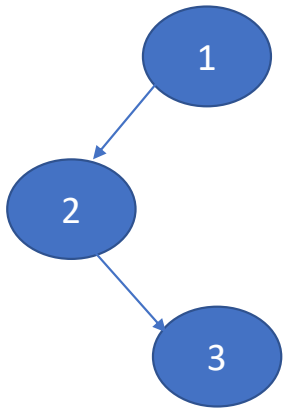
```
void postorder(struct node *root) {
  if (root == NULL) {
    return;
  }
  postorder(root->left);
  postorder(root->right);
  printf("%d ", root->val);
}
```

Exercise

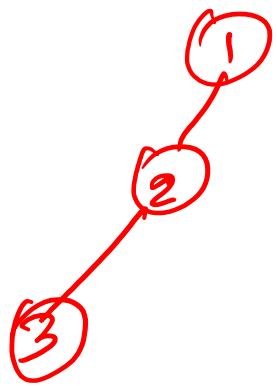
- Write all intermediate steps during the preorder, inorder, and postorder traversal of a binary tree that has one, five, or ten nodes

Reconstructing binary tree from preorder

- Given preorder traversal, can we reconstruct the original binary tree

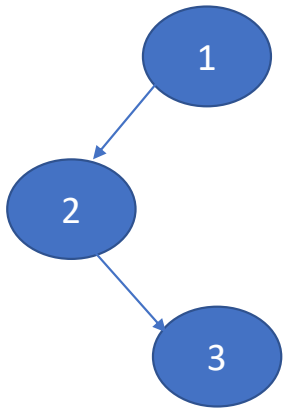


1 2 3

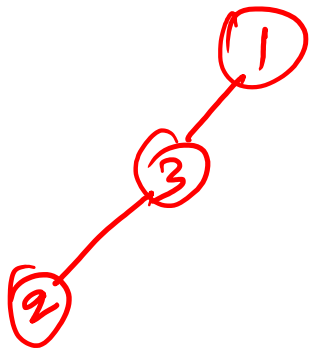


Reconstructing binary tree from inorder

- Given inorder traversal, can we reconstruct the original binary tree



2 3 1



Exercise

- Write an algorithm to reconstruct the unique tree from given inorder and preorder traversals (assuming no duplicates)

- Write an algorithm to reconstruct the unique tree from given inorder and postorder traversals (assuming no duplicates)

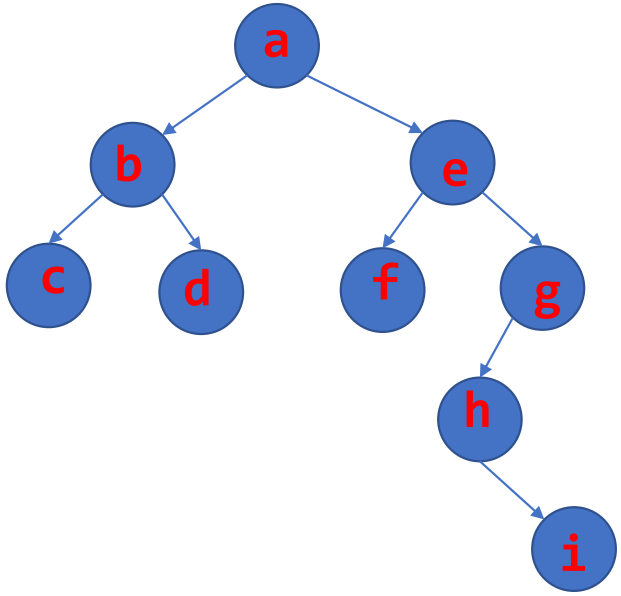
- Give an example to demonstrate that we can’t reconstruct a unique tree from given preorder and postorder traversals (assuming no duplicates)

Tree algorithms

Height of a binary tree

height (root)

1 + max (height (root → left),
height (root → right));



Height of a binary tree

- Recursive step
  - `h_left` = Recursively calculate the height of the left subtree
  - `h_right` = Recursively calculate the height of the right subtree
  - the height of the tree is `1 + max(h_left, h_right)`

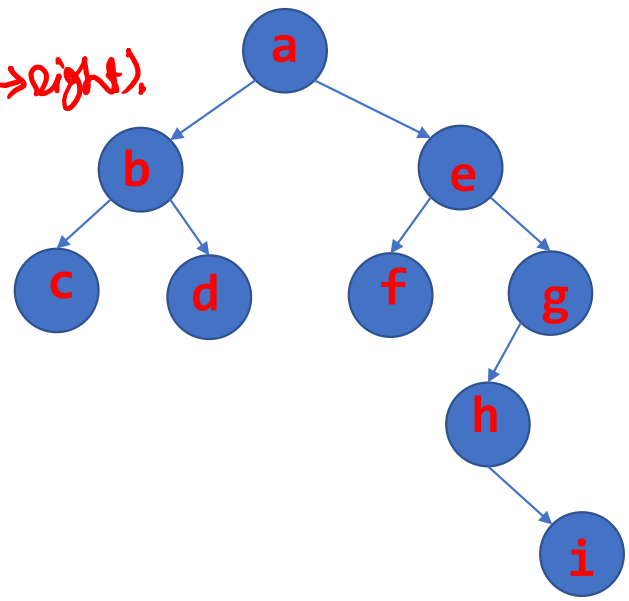
- Base case
  - if the tree is empty, `return 0`

Height of a binary tree

```
int height(struct node *root)
{
  if (root == NULL) {
    return 0;
  }
  return 1 + max(height(root->left), height(root->right));
}
```

Count the number of nodes

```
count (root)
{
    1 + count(root->left) + count(root->right).
}
```



Count the number of nodes

- Recursive step
  - `l_count` = recursively count the number of nodes in the left subtree
  - `r_count` = recursively count the number of nodes in the right subtree
  - The total number of nodes is `l_count + rcount + 1`

- Base case
  - If the tree is empty, `return 0`

Count the number of nodes

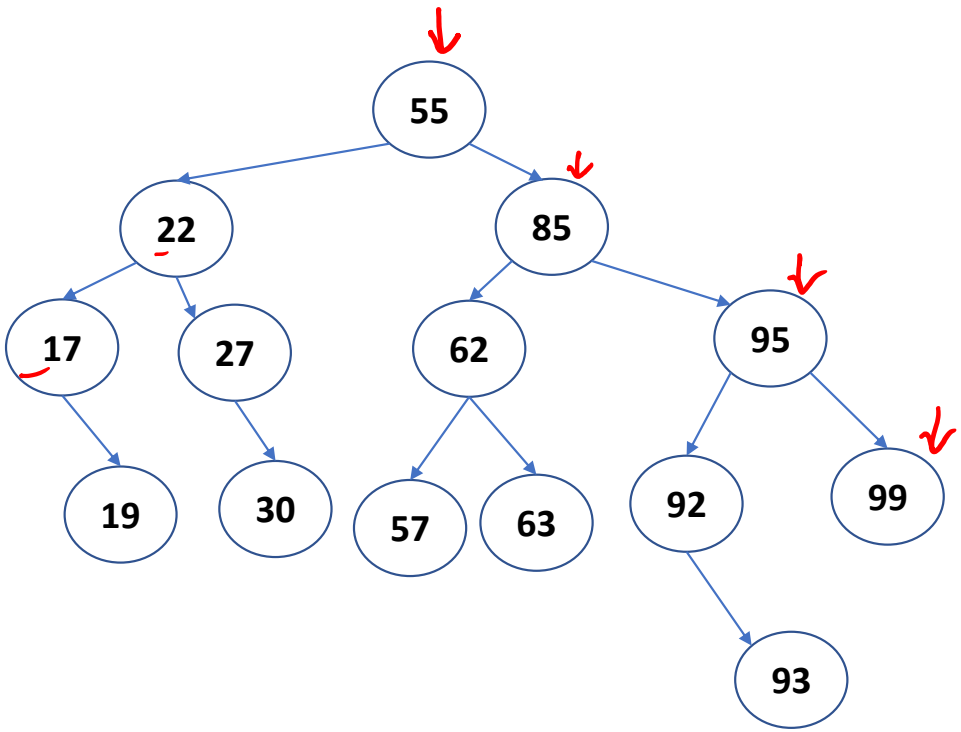
```
int count(struct node *root) {
  if (root == NULL) {
    return 0;
  }
  return 1 + count(root->left) + count(root->right);
}
```

Binary search tree

Binary search tree (BST)

- BST is a binary tree
- The value stored at the root node is more than the values stored in all the nodes in the left subtree
- The value stored at the root node is less than or equal to the values stored in all the nodes in the right subtree
- The empty tree (NULL) is a BST
- The left and right subtrees in a BST are BSTs

BST



The leftmost element is the smallest element.
The rightmost element is the largest element.

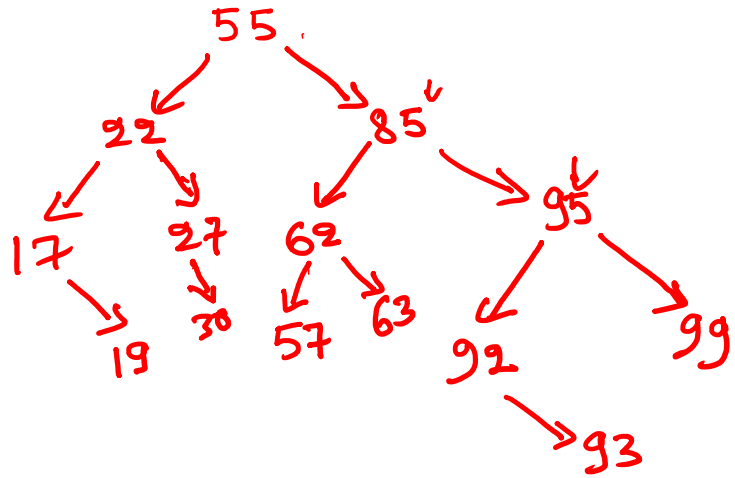
Insertion

Insertion

- The `insert` routine inserts a new node in the BST with a given value

BST insertion

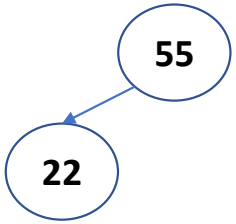
55 22 85 95 92 17 99 62 93 19 27 57 63 30



BST insertion

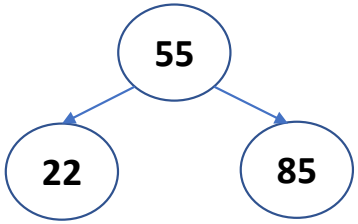
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



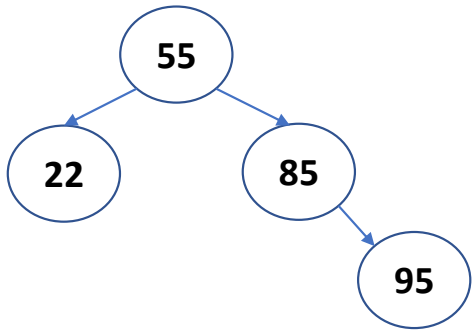
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



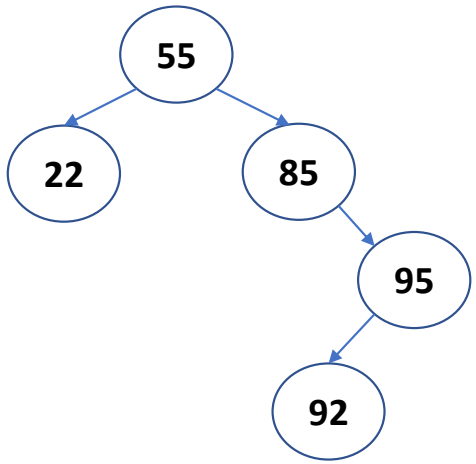
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



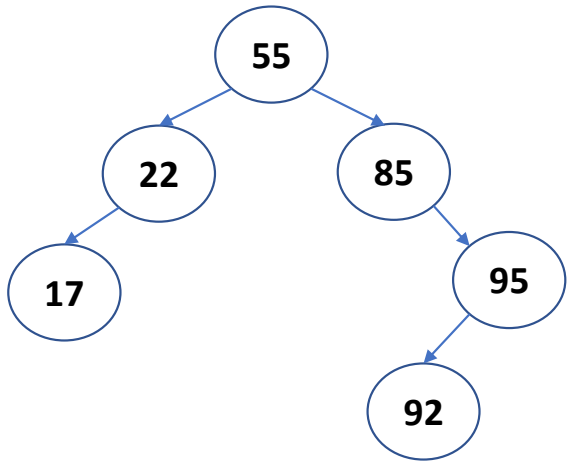
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



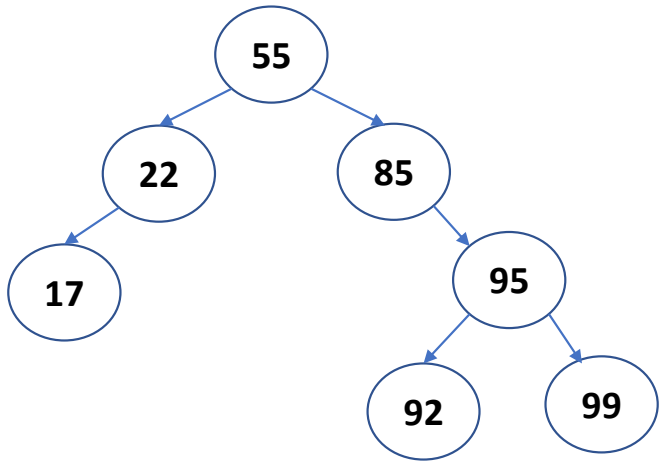
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



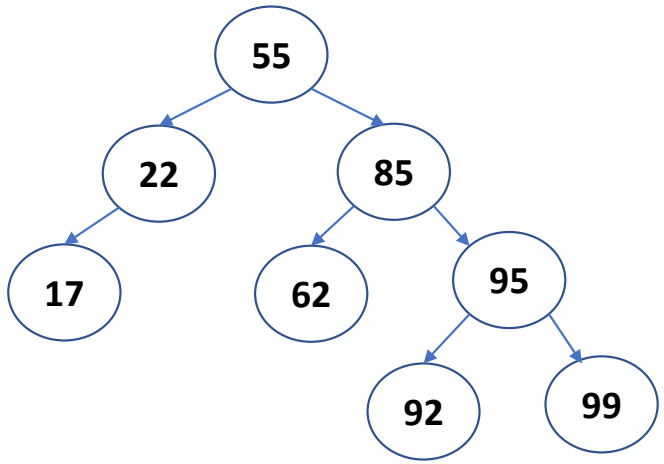
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



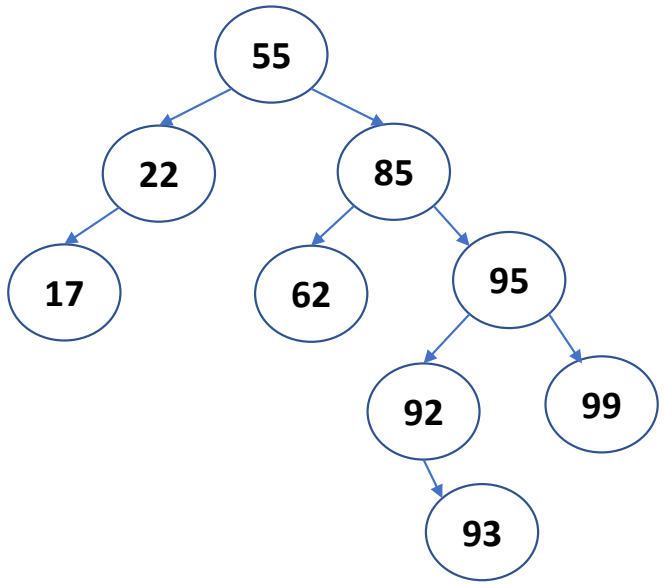
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



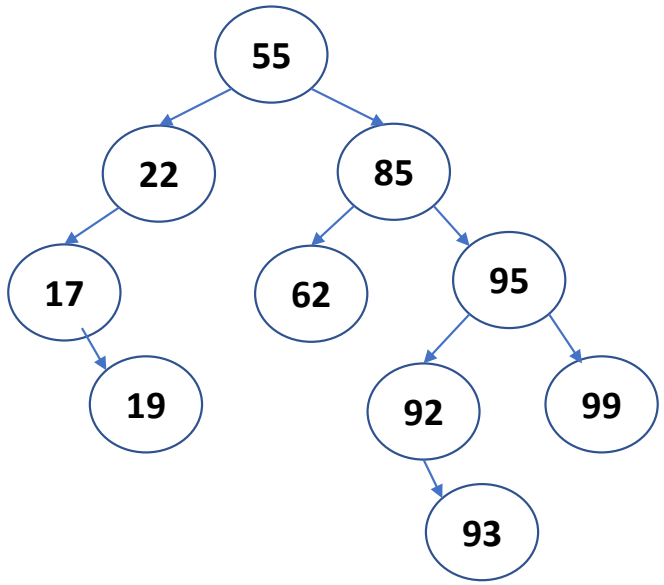
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



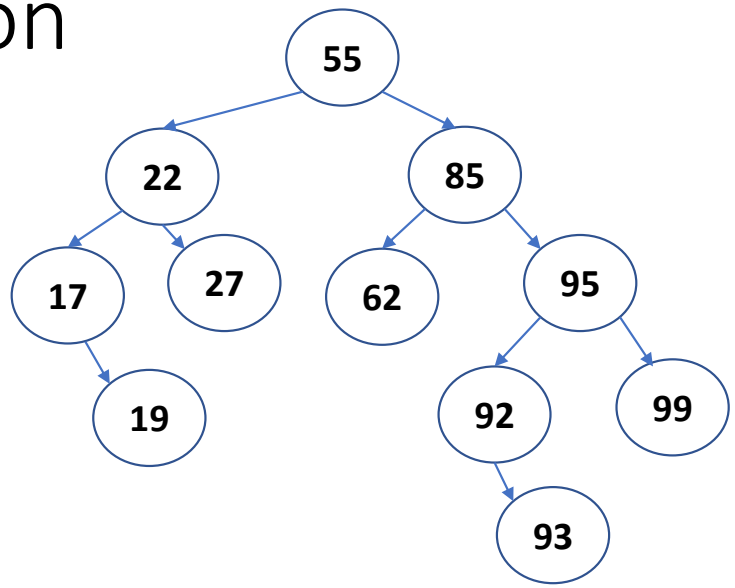
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



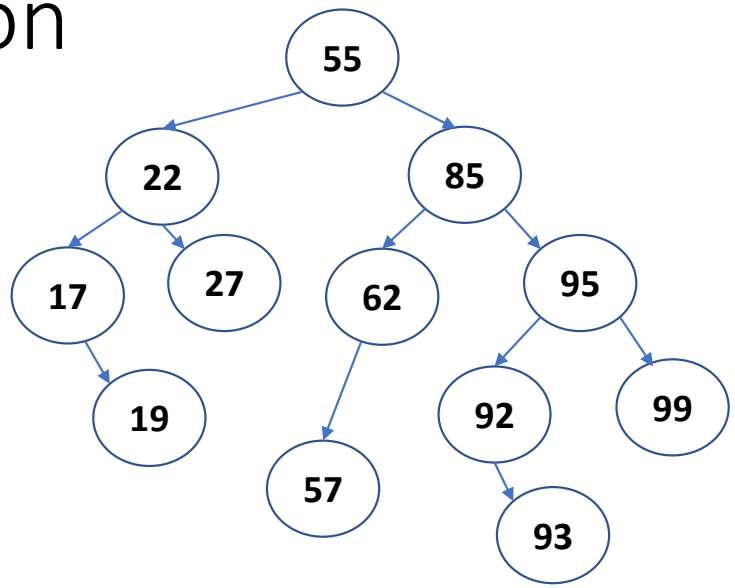
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



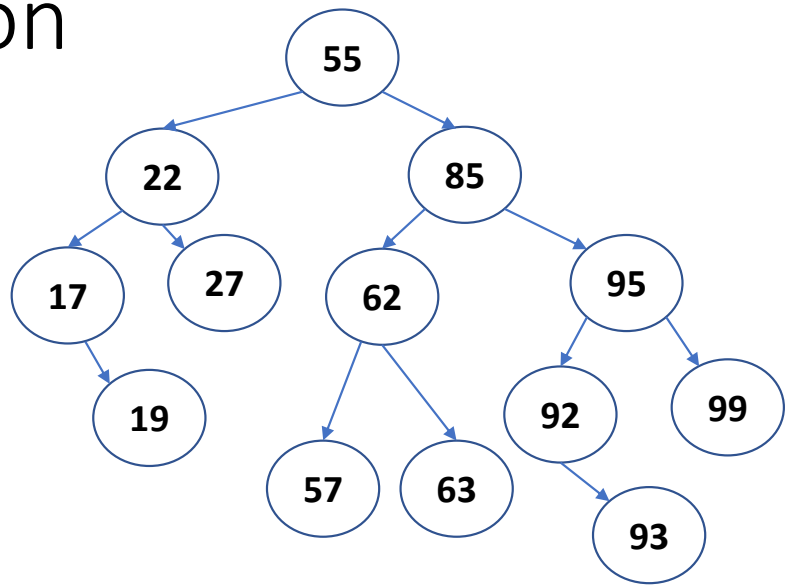
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



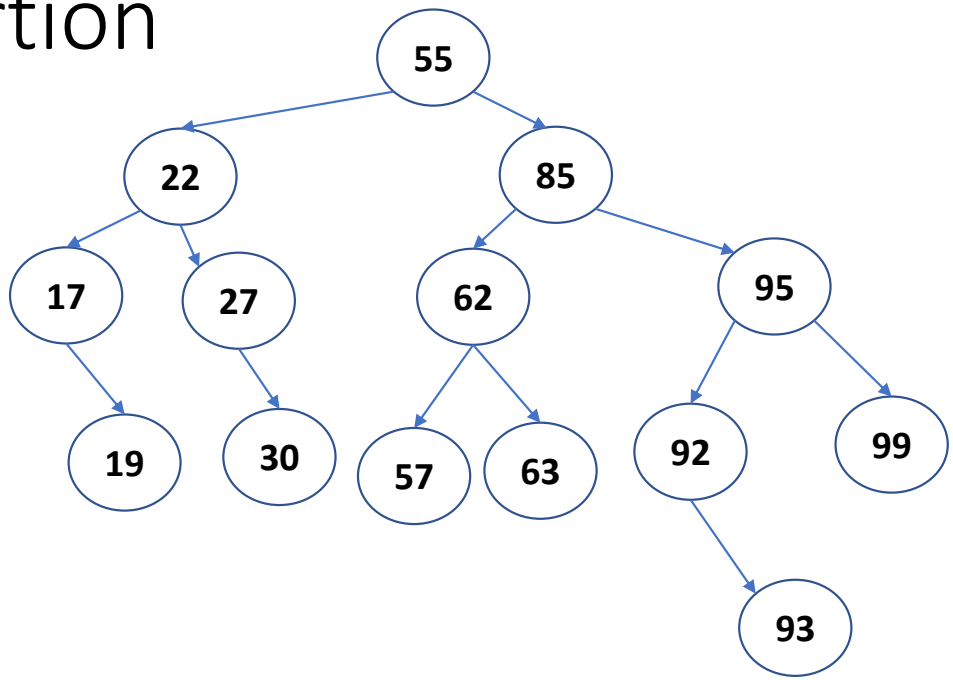
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion (recursive algorithm)

- `insert` returns the root of the tree after inserting a node with value (`val`)
- Recursive step

```
if ( val < root->val)
    root->left = insert(root->left, val)
else
    root->right = insert(root->right, val);
```

- Base step

```
allocate_node (val);
```

BST insertion (recursive algorithm)

- `insert` returns the root of the tree after inserting a node with value (`val`)
- Recursive step
  - if the `val` is greater than or equal to the value stored at `root`
    - recursively insert in the right subtree
    - make root point to the new root of the right subtree
  - if the `val` is less than the value stored at `root`
    - recursively insert in the left subtree
    - make root point to the new root of the left subtree
- Base step
  - if the `root` is `null,` allocate and return a new node that stores `val`

BST insertion

```
struct node* insert(struct node *root, int val) {

  if (root == NULL) {
    /* allocate_node sets the left and right fields to NULL */
    root = allocate_node(val);
    return root;
  }

  if (val >= root->val) {
    root->right = insert(root->right, val);
  }
  else {
    root->left = insert(root->left, val);
  }
  return root;
}
```

Time complexity

- Complete binary tree $T(n) = T(n/2) + c \simeq O(\log n)$

- Skewed tree

$O(n)$

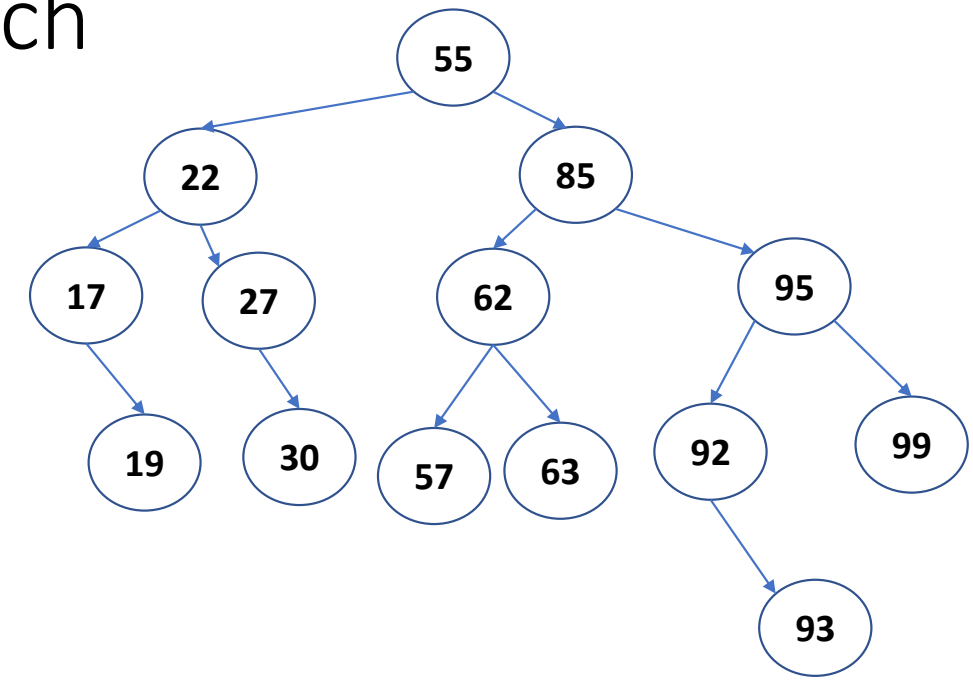
Search

Search

- The `search` routine tries to find a node in the BST that stores a given input value

- If none of the nodes stores the value, `search` returns `NULL`

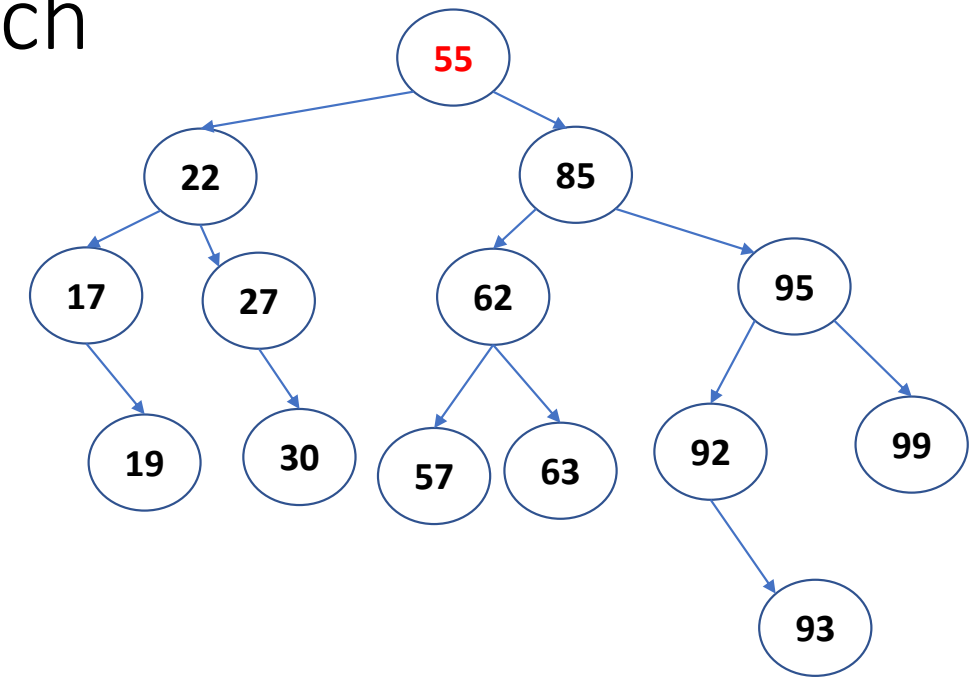
BST search



search 93

55 22 85 95 92 17 99 62 93 19 27 57 63 30

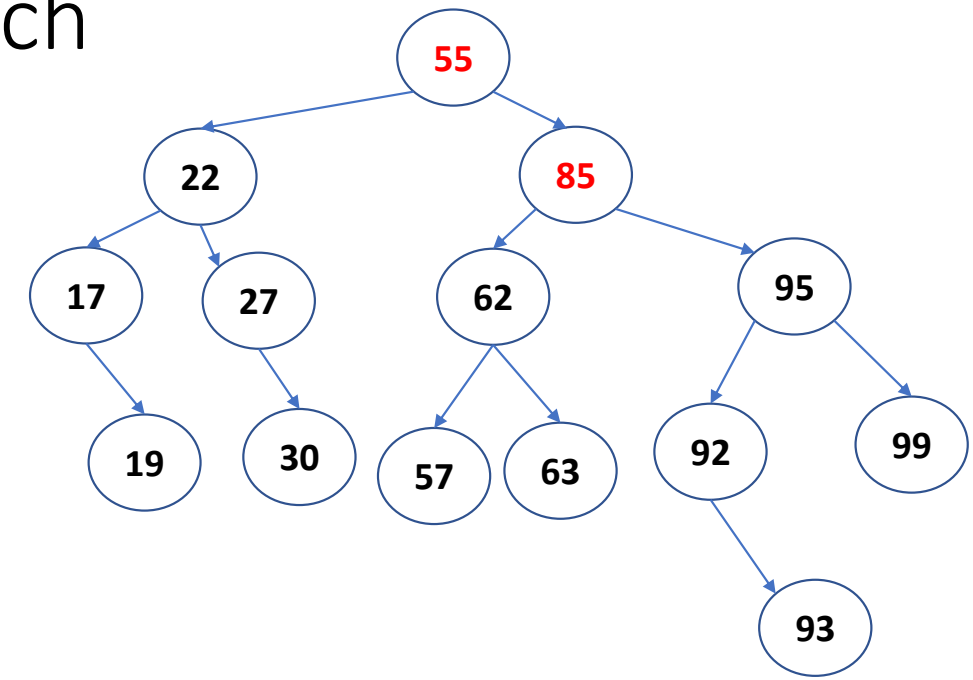
BST search



search 93

55 22 85 95 92 17 99 62 93 19 27 57 63 30

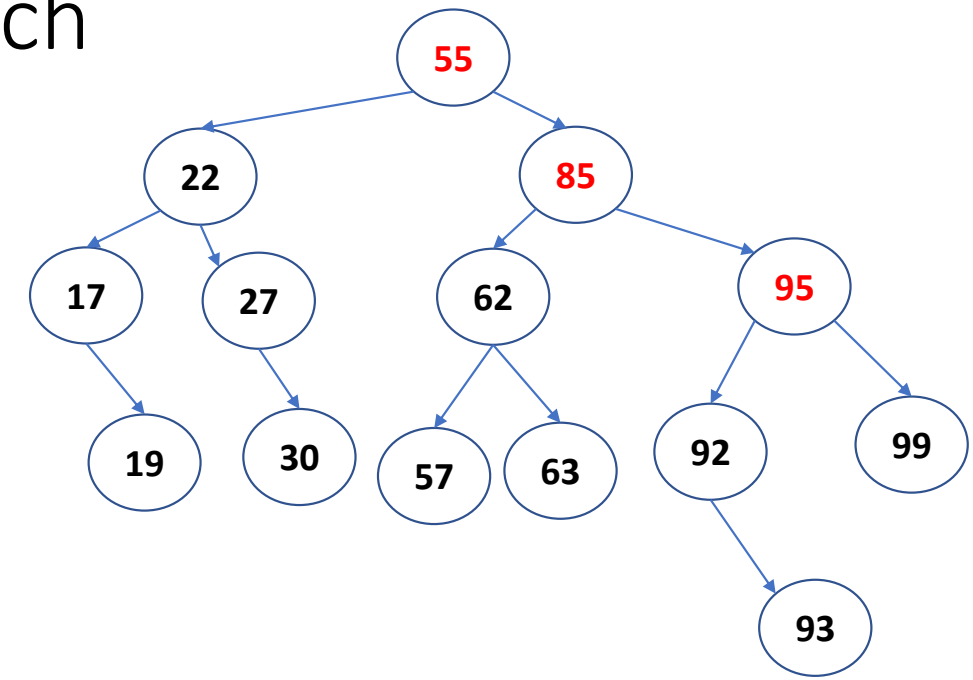
BST search



search 93

55 22 85 95 92 17 99 62 93 19 27 57 63 30

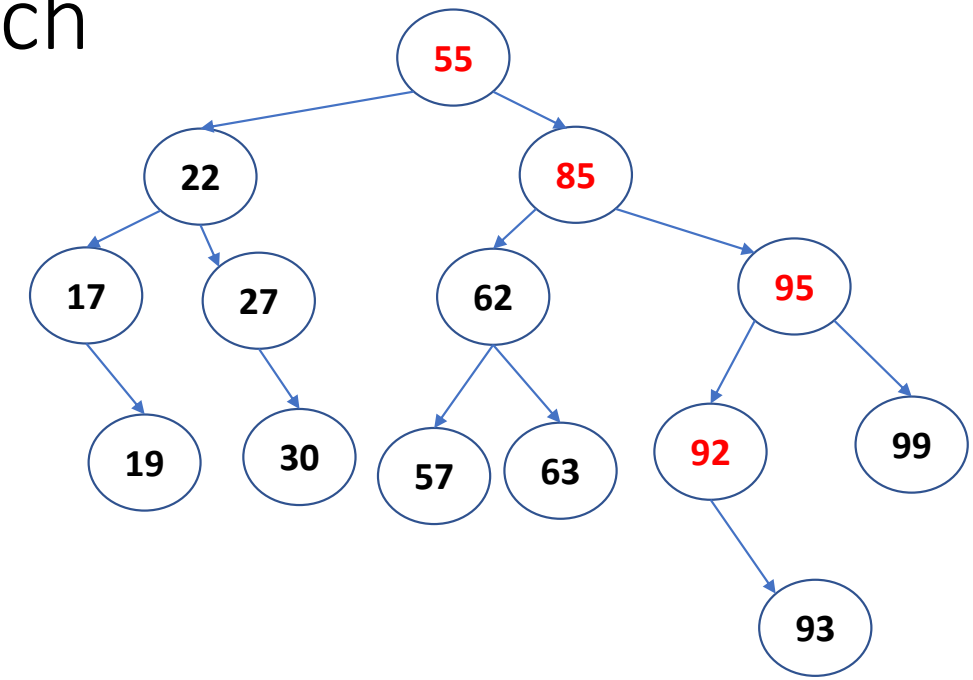
BST search



search 93

55 22 85 95 92 17 99 62 93 19 27 57 63 30

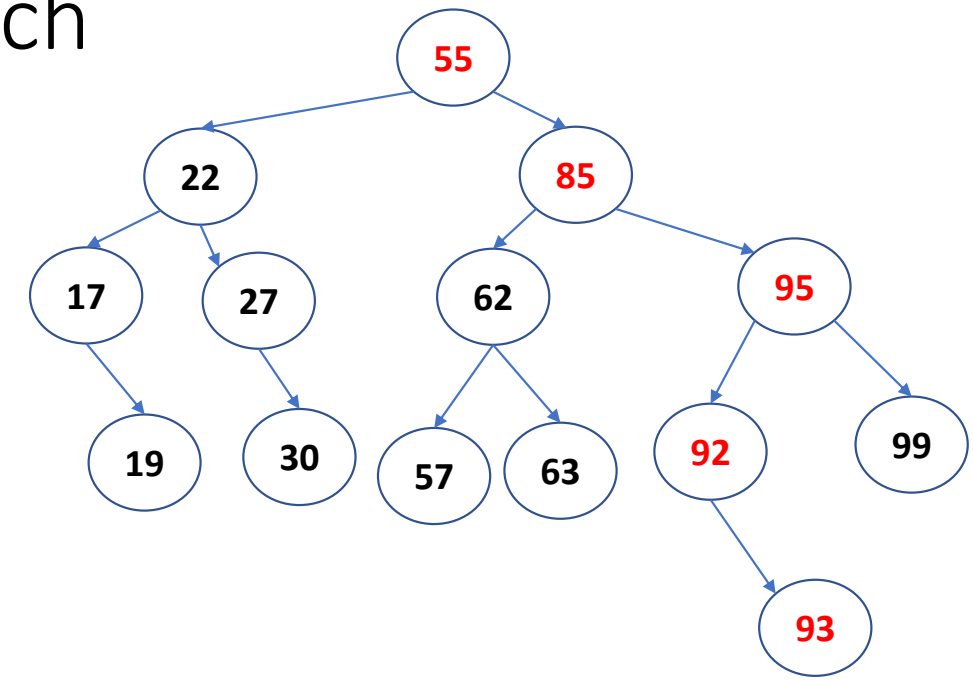
BST search



search 93

55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST search



search 93

55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST search (recursive algorithm)

- `search` returns a node that stores the value (`val`) being searched
- if `val` is not stored in any of the nodes, `search` returns `NULL`
- Recursive step

```
search(root)
    if (val > root->val) {
        return search(root->right);
    }
    else if (val < root->val) {
        return search(root->left);
    }
```

- Base step

```
if (root == NULL)
    return NULL;
if (root->val == val)
    return root;
```

BST search (recursive algorithm)

- `search` returns a node that stores the value (`val`) being searched
- If `val` is not stored in any of the nodes, `search` returns `NULL`
- Recursive step
  - if `val` is greater than the value stored at `root`
    - recursively search in the right subtree
  - if `val` is less than the value stored at `root`
    - recursively search in the left subtree
- Base step
  - if the `root` is `NULL`, return `NULL`
  - If the value stored at `root` is `val`, `return root`

BST search

```
struct node *search(struct node *root, int val) {
  if (root == NULL) {
    return NULL;
  }
  if (root->val == val) {
    return root;
  }
  else if (val > root->val) {
    return search(root->right, val);
  }
  else {
    return search(root->left, val);
  }
}
```

Time complexity

- Complete binary tree $O(\log n)$
- Skewed tree $O(n)$

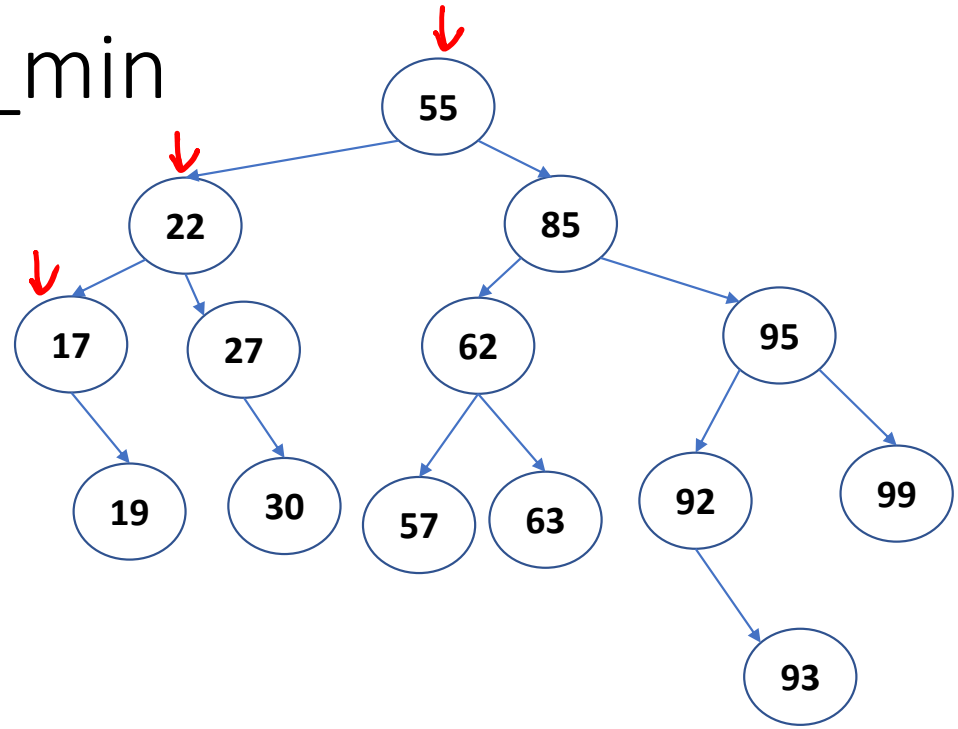
Finding minimum node

Finding minimum node (find_min)

- The goal of `find_min` is to find a node that stores the minimum value in a BST

- The leftmost element in a BST contains the minimum value

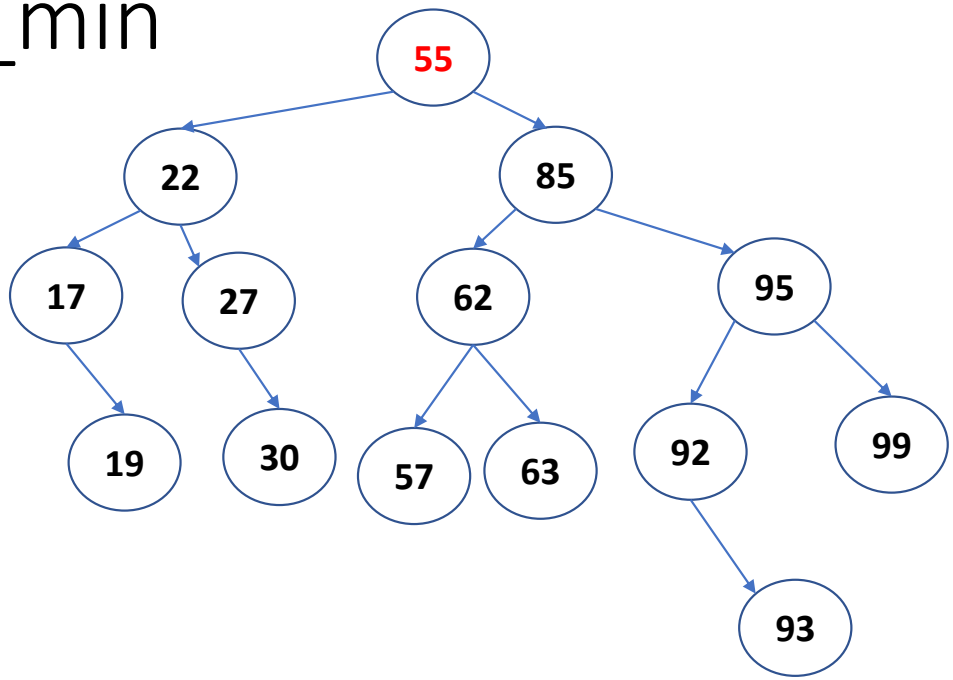
BST find_min



17

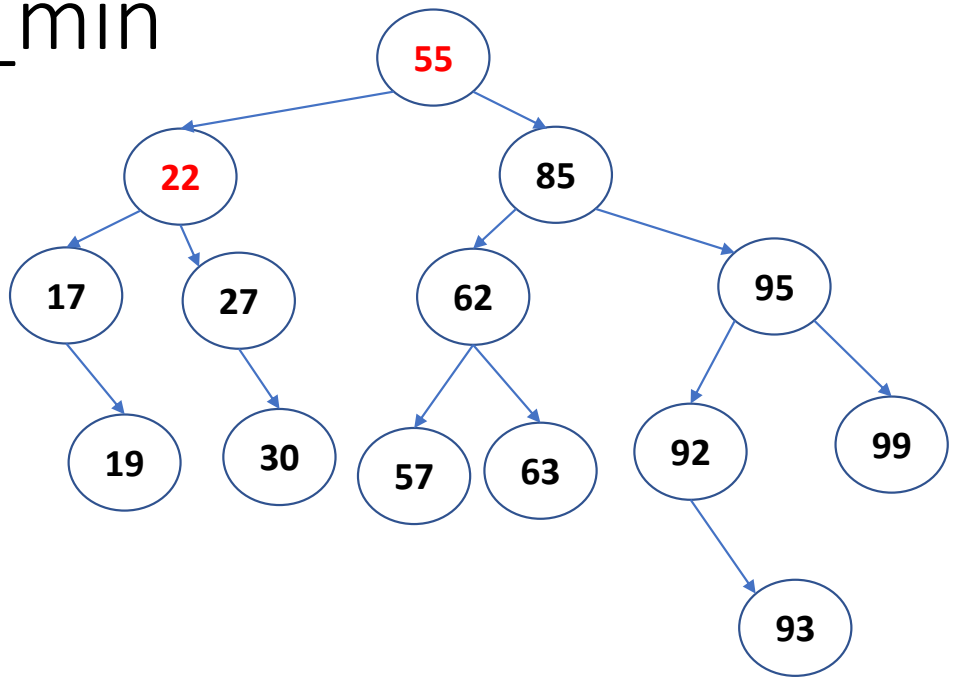
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST find_min



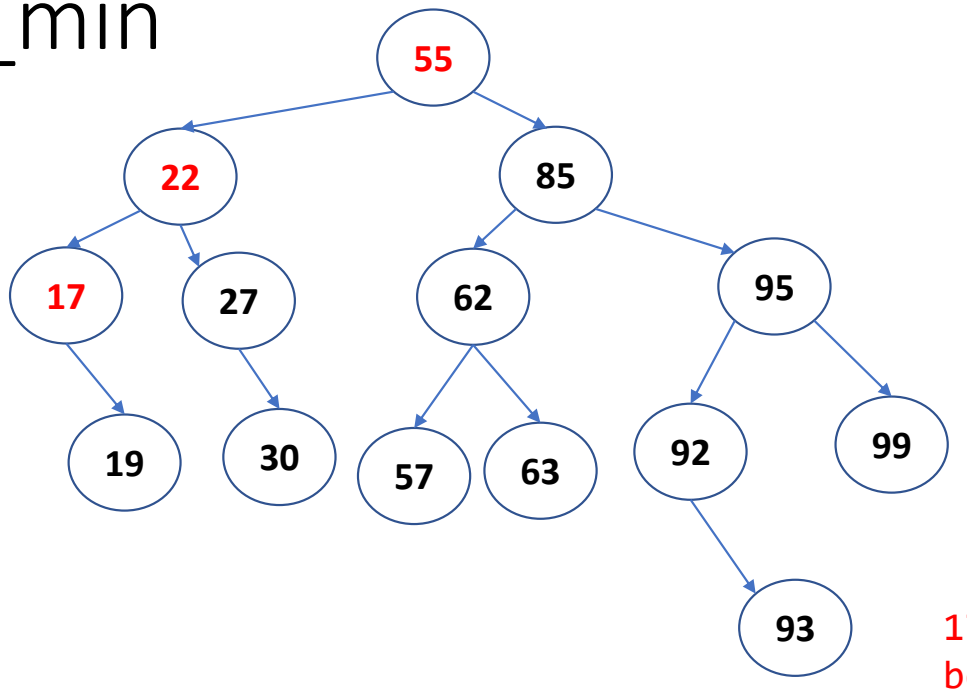
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST find_min



55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST find_min



17 is the minimum node because it has no left subtree.

55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST find_min

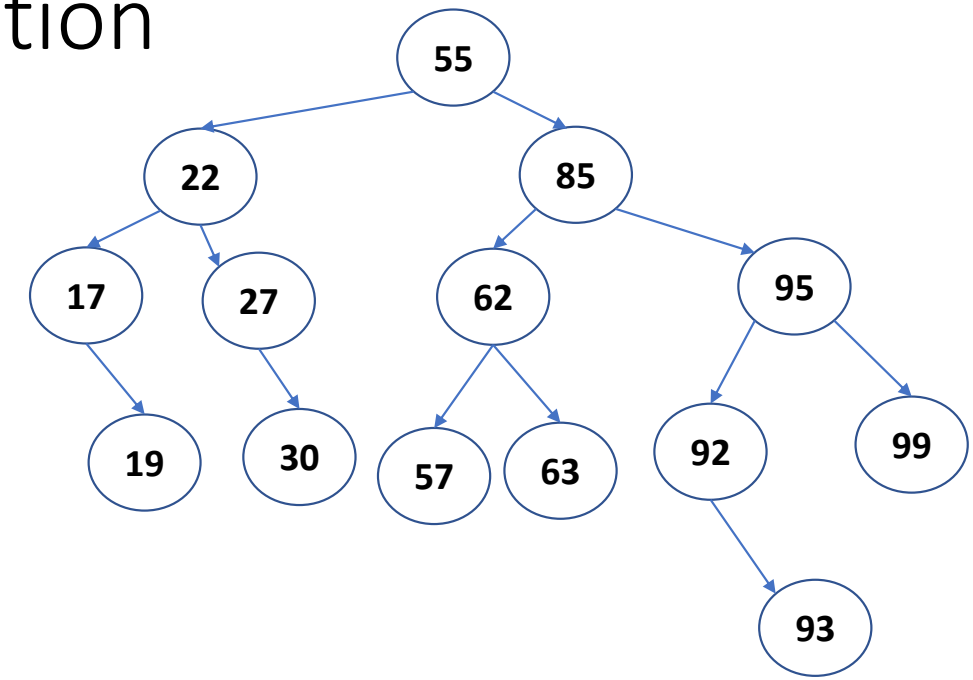
```
// assumes that the input tree is not empty

struct node* find_min(struct node *root) {
  if (root->left == NULL) {
    return root;
  }
  else {
    return find_min(root->left);
  }
}
```

Deletion

Delete

- The goal of the delete operation is to delete a node from a BST that stores a given input value

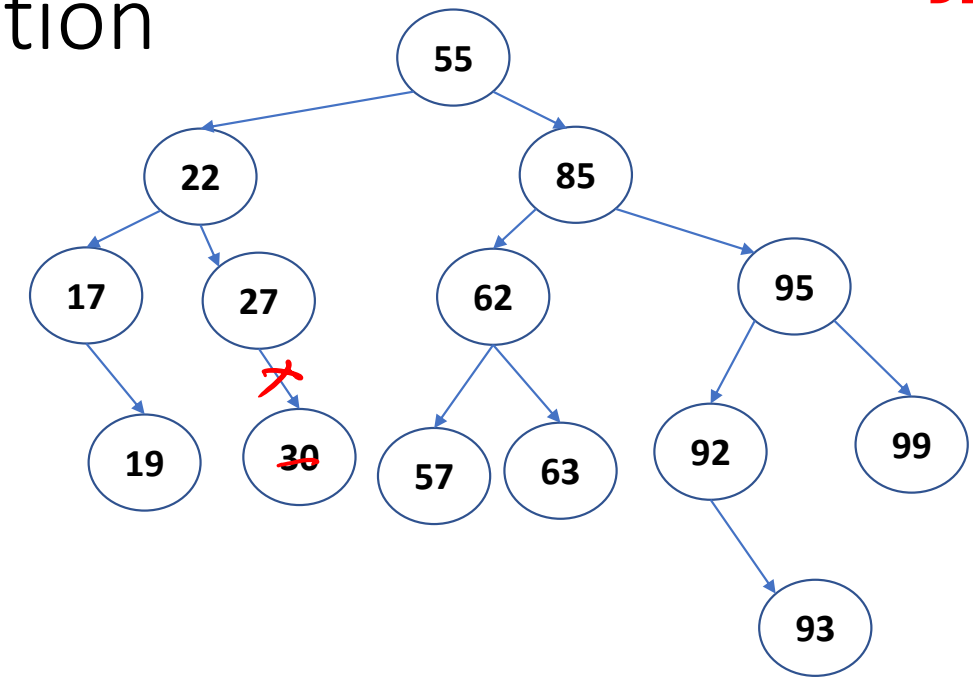
BST deletion



55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST deletion

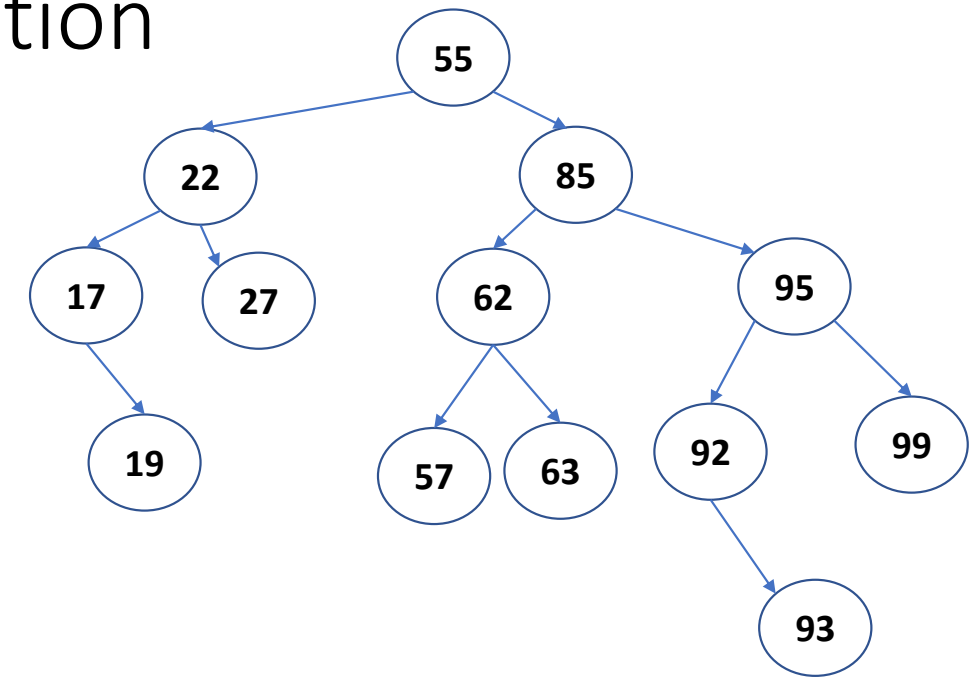
DELETING A LEAF NODE



delete 30

55 22 85 95 92 17 99 62 93 19 27 57 63 30

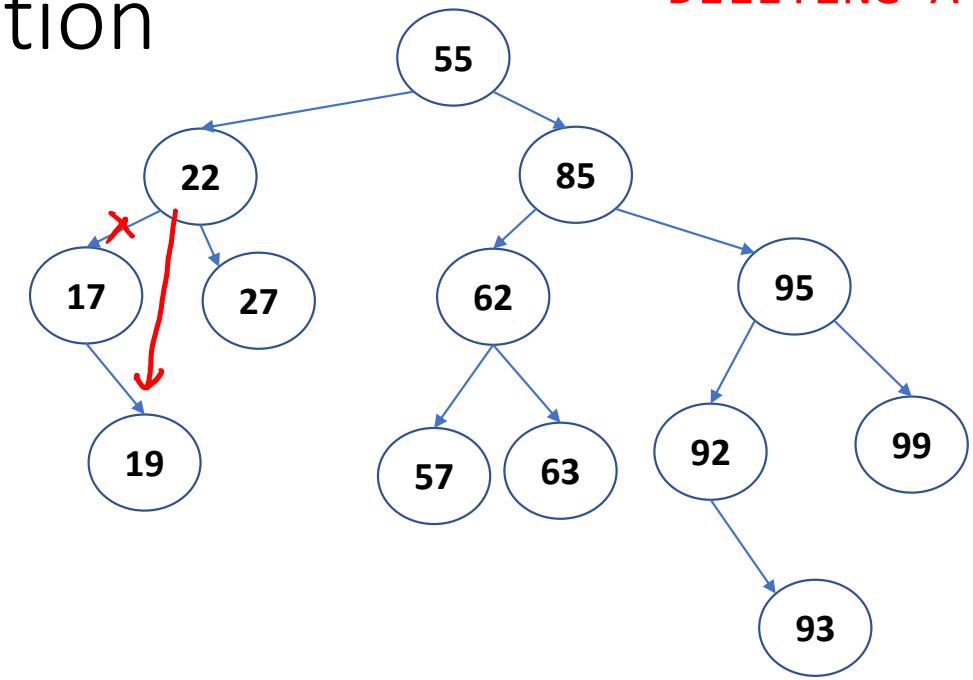
BST deletion



55 22 85 95 92 17 99 62 93 19 27 57 63 ~~30~~

BST deletion

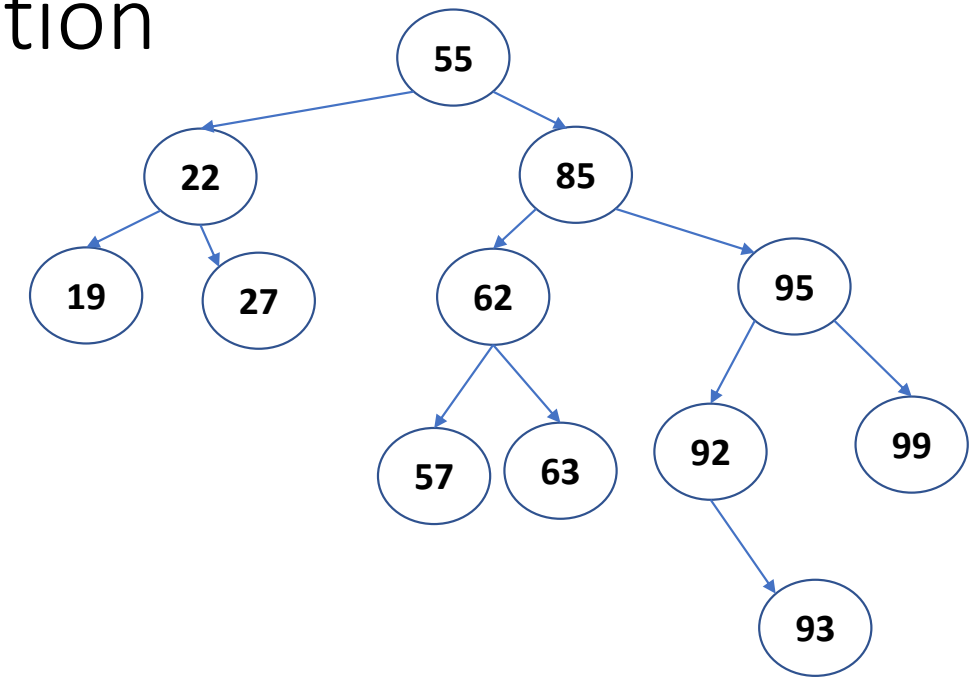
DELETING A NODE WITH ONE CHILD



delete 17

55 22 85 95 92 17 99 62 93 19 27 57 63 ~~30~~

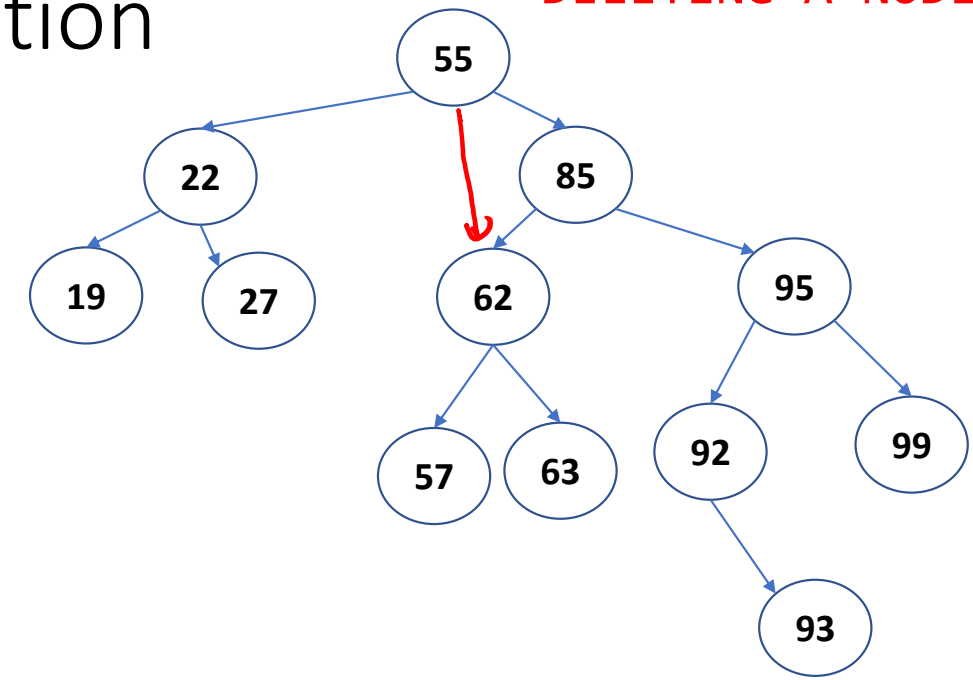
BST deletion



55 22 85 95 92 ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion

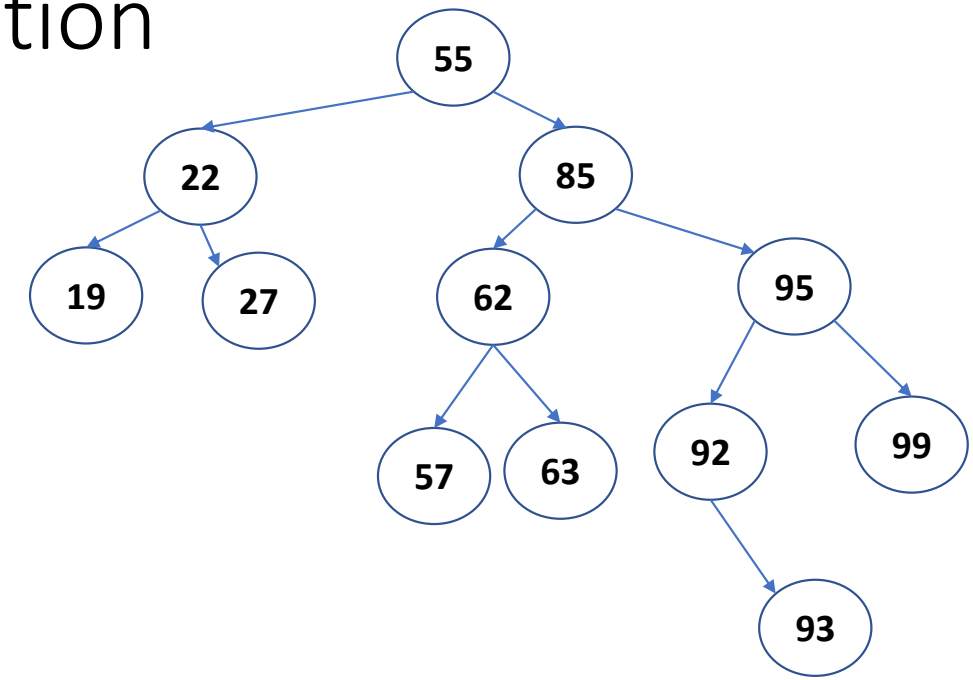
DELETING A NODE WITH TWO CHILDREN



delete 85

55 22 85 95 92 ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion

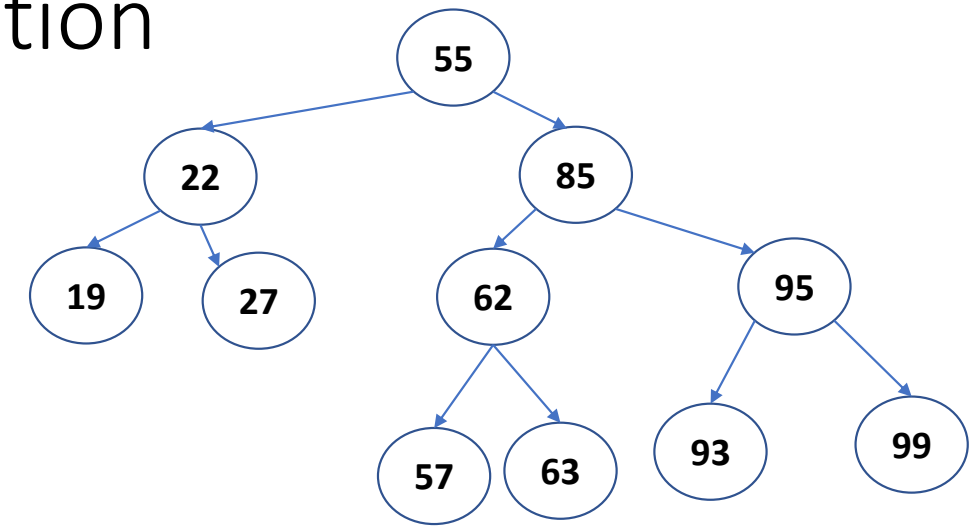


delete 85

delete a node with the minimum value in the right subtree (i.e., 92)

55 22 85 95 92 ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion

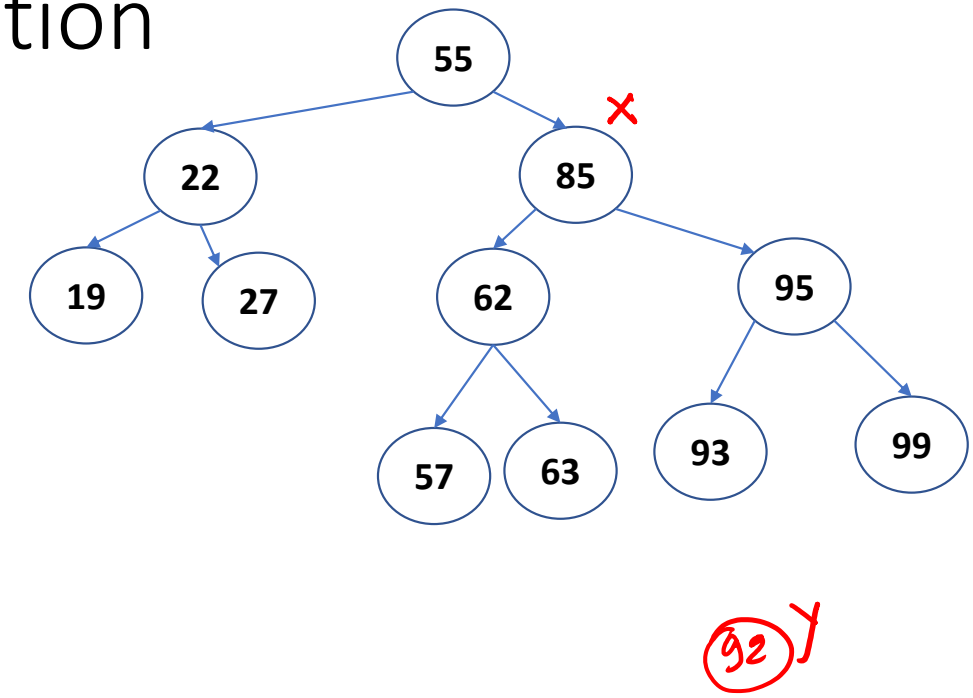


delete 85

delete a node with the minimum value in the right subtree (i.e., 92)

55 22 85 95 ~~92~~ ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion



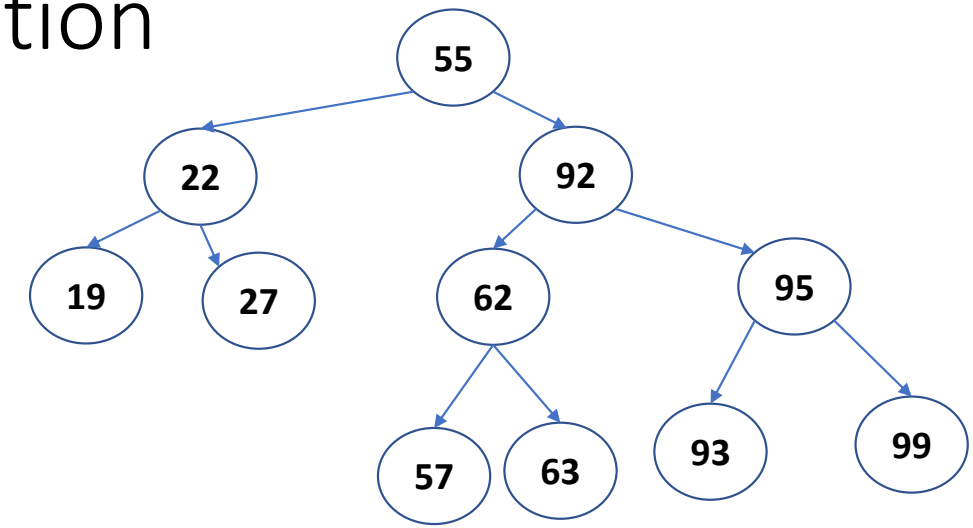
delete 85

delete a node with the minimum value in the right subtree (i.e., 92)

replace 85 with 92

55 22 85 95 ~~92~~ ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion



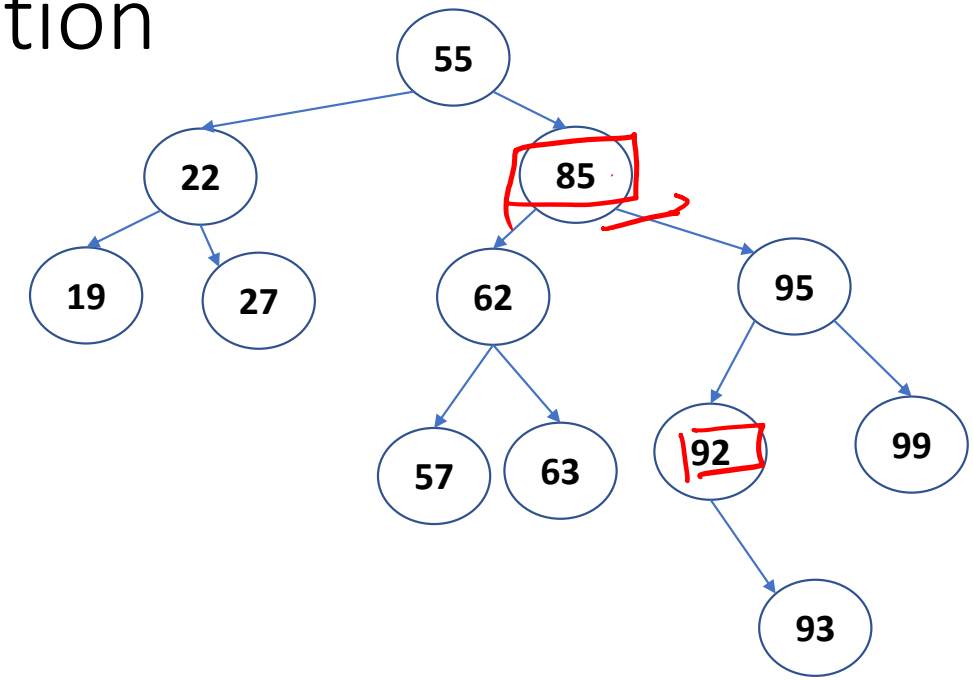
delete 85

delete a node with the minimum value in the right subtree (i.e., 92)

replace 85 with 92

55 22 ~~85~~ 95 ~~92~~ ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion



delete 85

replace the node with the minimum value node in the right subtree (i.e., 92)

What is the other alternative?

replace the node with the maximum value node in the left subtree (i.e., 63)

55 22 85 95 92 ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion (recursive algorithm)

- `delete` returns the root of the tree after deleting a node that contains a given value (`val`)
- Recursive step

```
if (val < root->val)
    root->left = delete( root->left);
else if (val > root->val)
    root->right = delete(root->right);
else if (val == root->val && root->right != NULL && root->left != NULL
{
    min = find_min(root->right);
    root->val = min->val;
    delete (root->right, min->val);
```

- Base step

BST deletion (recursive algorithm)

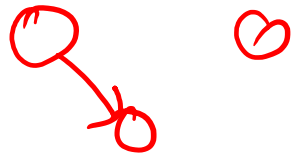
- `delete` returns the root of the tree after deleting a node that contains an input value (`val`)
- Recursive step
  - if `val` is greater than the value stored at `root`, recursively delete from the right subtree
    - make root point to the new root of the right subtree
  - if `val` is less than the value stored at `root`, recursively delete from the left subtree
    - make root point to the new root of the left subtree
  - if `val` is stored at `root` and `root` has both left and right children, recursively delete the node with the minimum value (say `x`) from the right subtree and store `x` in `root`

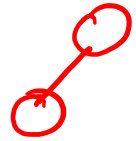
BST deletion (recursive algorithm)

- Base step
  - if `root` is `null,` allocate and return a new node with value `val`
  - if `val` is stored at `root` and `root` has at most one child
    - if `root` is a leaf node, return empty tree
    - if `root` is not leaf, return the child node

BST deletion

```
struct node* delete(struct node *root, int val) {
  if (root == NULL) {
    return NULL;
  }
  if (root->val == val) {
    if (root->left == NULL) {
      struct node *ret = root->right;
      free(ret);
      return ret;
    }
    else if (root->right == NULL) {
      struct node *ret = root->left;
      free(ret);
      return ret;
    }
    else if (root->right != NULL && root->left != NULL) {
      struct node *min_node = find_min(root->right);
      root->val = min_node->val;
      root->right = delete(root->right, min_node->val);
      return root;
    }
  }
  else if (val > root->val) {
    root->right = delete(root->right, val);
  }
  else {
    root->left = delete(root->left, val);
  }
  return root;
}
```





Time complexity

- Complete binary tree $O(\log n)$
- Skewed tree $O(n)$

Today’s topics

- AVL tree

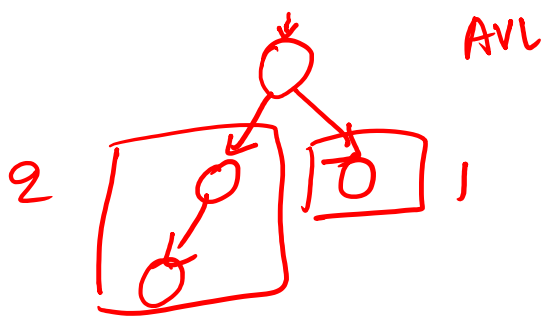
References

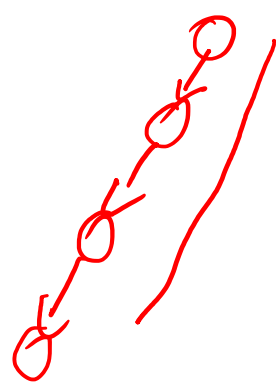
- Section-10.2 from Goodrich et al.

AVL tree

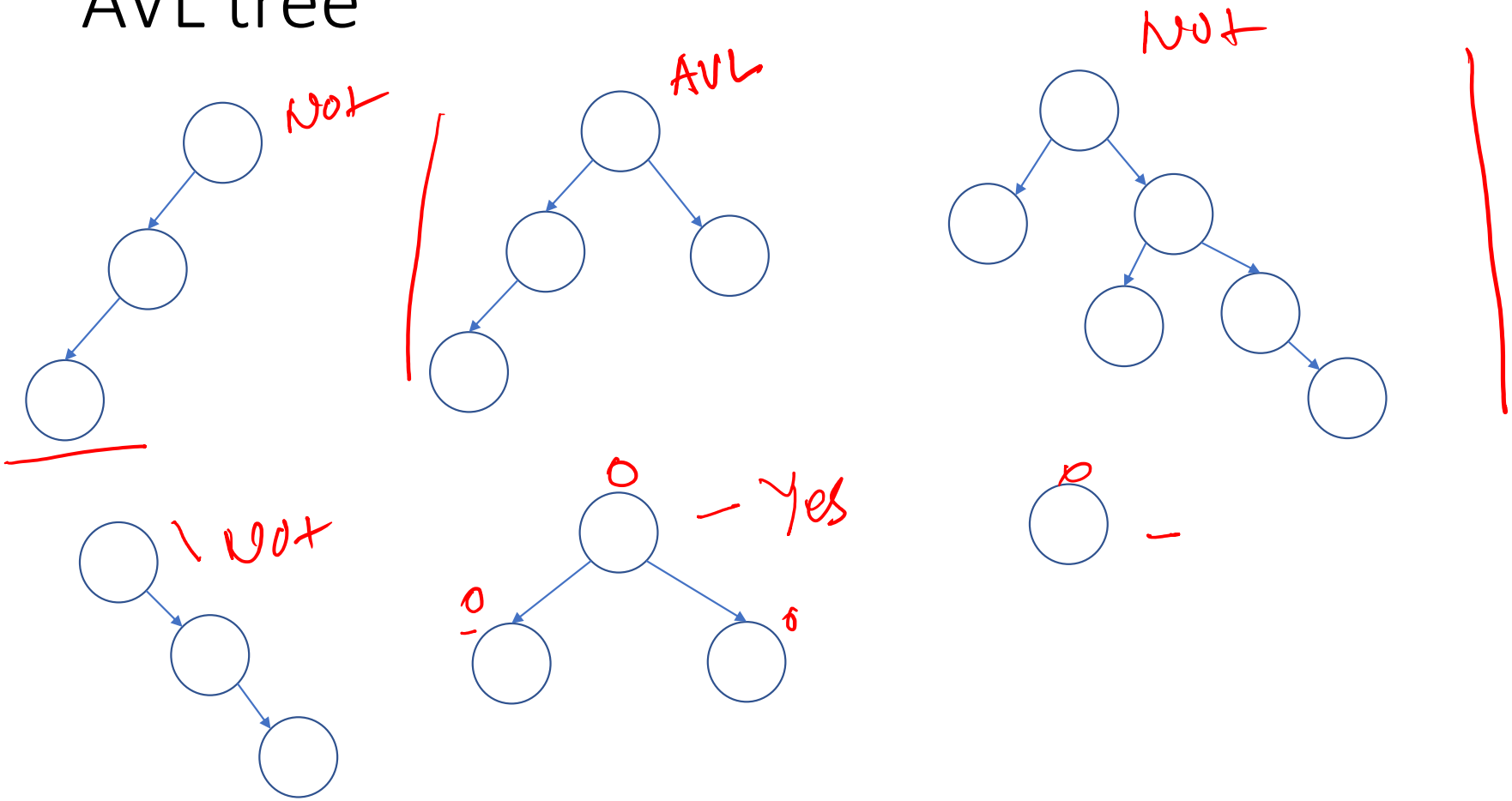
- AVL tree is a BST that satisfies the **height-balance** property, and nodes that store equal values can be present on both sides

- height-balance property: For each node in the tree, the heights of the left and right subtrees differ by at most one





AVL tree



AVL tree height

- What is the lower bound for the height of an AVL tree with n nodes?

$\log_2(n+1)$

AVL tree height (upper bound)

- Let's say f(h) represents the minimum number of nodes in an AVL tree of height h

$$f(h) = f(h-1) + f(h-2) + 1$$

$$f(h) \geq 2f(h-2)$$
$$\geq 2^2 f(h-4)$$
$$\geq 2^3 f(h-6)$$
$$\cdots$$
$$\geq 2^k f(h-2k)$$

Substitute $k = \frac{h-1}{2}$

$$\geq 2^{\frac{h-1}{2}} f(1)$$

$$f(1) = 1$$
$$f(2) = 2$$

$$f(h) \geq 2^{\frac{h-1}{2}}$$

$$\frac{h-1}{2} \leq \log_2 f(h)$$

$$h \leq 2\log_2 f(h) + 1$$

$$n \geq f(h)$$

$$h \leq 2\log_2 n + 1$$

AVL tree height (upper bound)

- Let's say f(h) represents the minimum number of nodes in an AVL tree

$f(1) = 1, f(2) = 2$

$f(h) = 1 + f(h-1) + f(h-2)$

$f(h) > 2f(h-2)$

$> 2^2 f(h-4)$

$> 2^3 f(h-6)$

…

$> 2^k f(h-2k)$

Substituting $k = (h-1)/2$

$f(h) > 2^{(h-1)/2} f(1)$

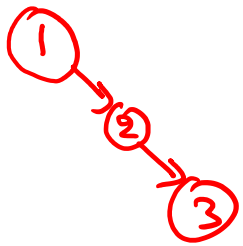
$h < 2\log_2 f(h) + 1 < 2\log_2 n + 1 = O(\log n)$

Rotations

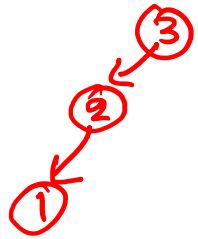
Rotations

Draw all BSTs with three nodes 1, 2, and 3.

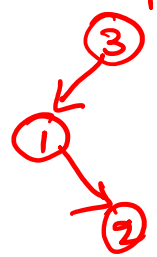
No



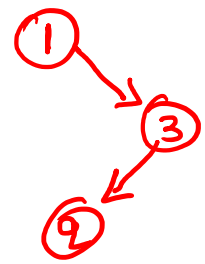
No



No



No

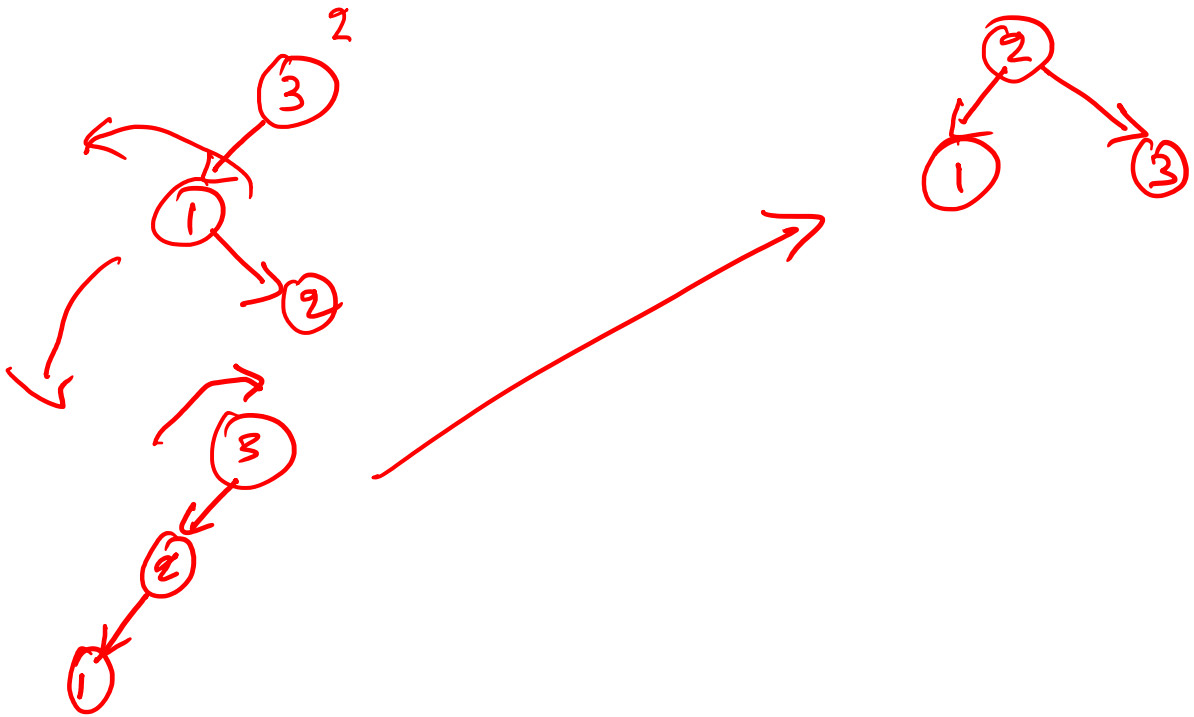


AVL

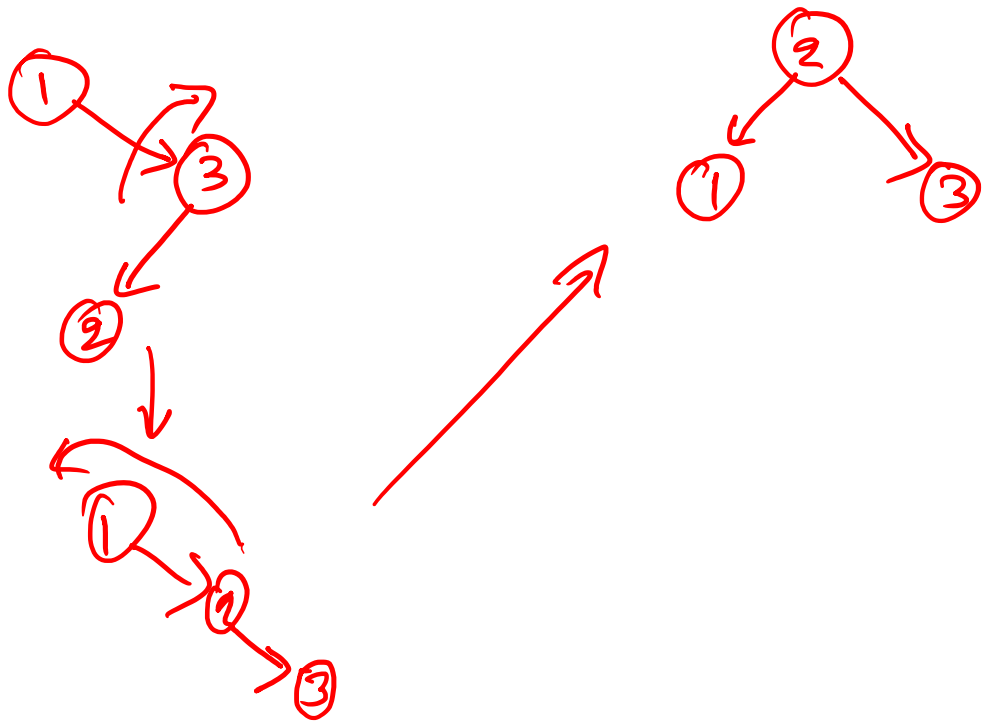
Single rotation

Single rotation

Double rotation



Double rotation

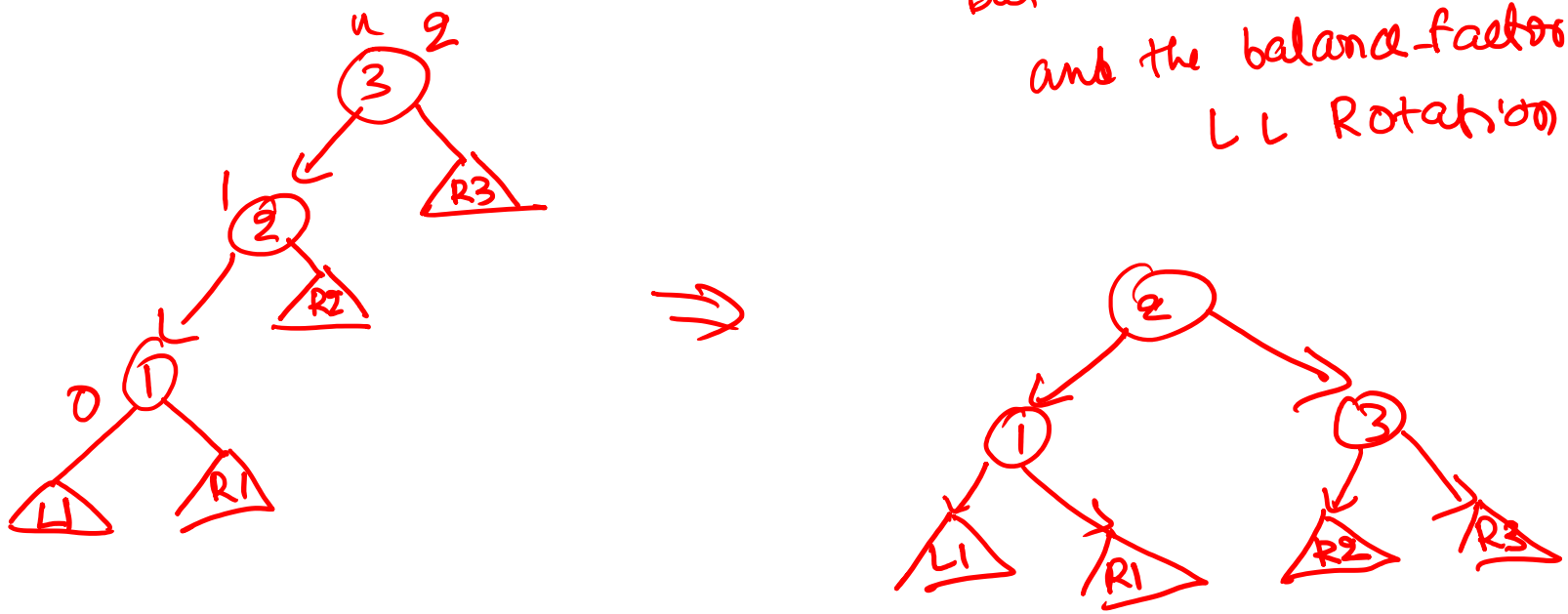


Balance factor

- **`balance_factor = height of left subtree – height of right subtree`**

LL rotation

balance_factor(n) > 1
and the balance_factor(n->left) ≥ 0
LL Rotation



Left heavy : balance factor > 0
Right heavy : balance factor < 0

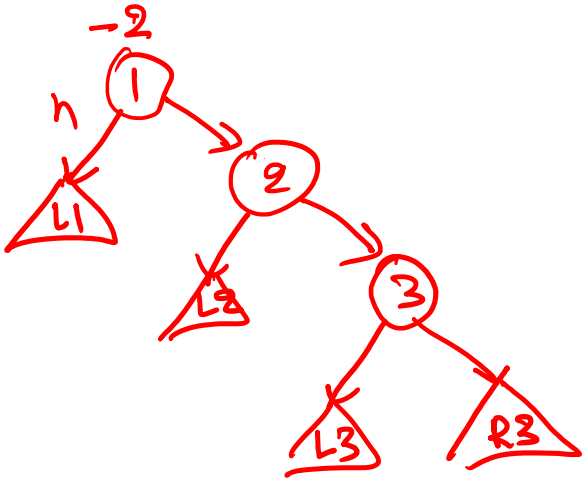
LL rotation

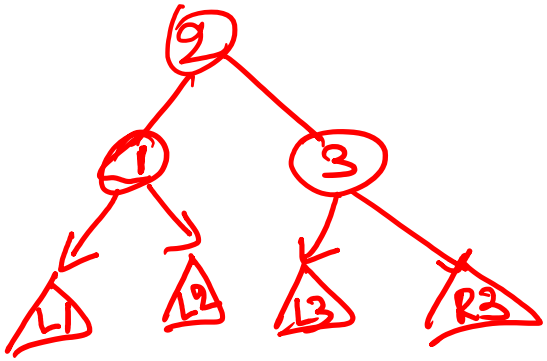
- We perform LL rotation on a node n, if
  - **balance_factor(n) > 1** and **n->left is not right heavy**

```
Left heavy : balance factor > 0
Right heavy : balance factor < 0
```

RR rotation

if balance factor(n) < −1
and balance_factor(n->right) ≤ 0





Left heavy : balance factor > 0
Right heavy : balance factor < 0

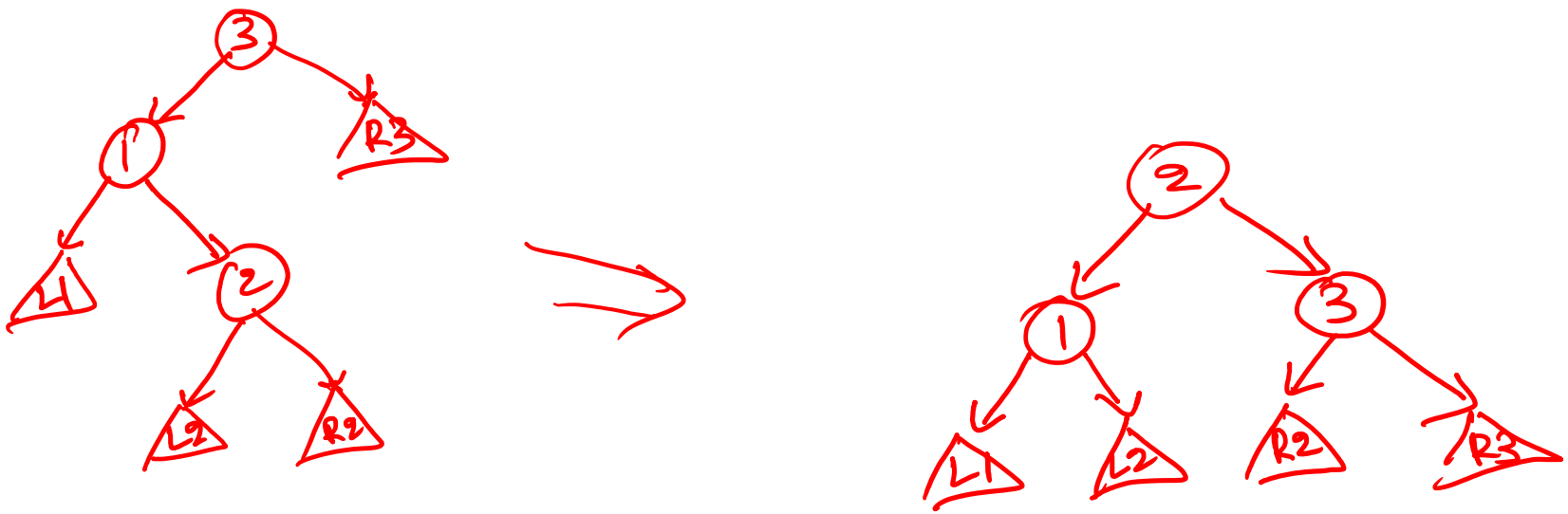
RR rotation

- We perform RR rotation on a node n, if
  - **balance_factor(n) < -1** and **n->right is not left heavy**

```
Left heavy : balance factor > 0
Right heavy : balance factor < 0
```

LR rotation

if balance_factor(n) > 1
and balance_factor(n->left) < 0



Left heavy : balance factor > 0
Right heavy : balance factor < 0

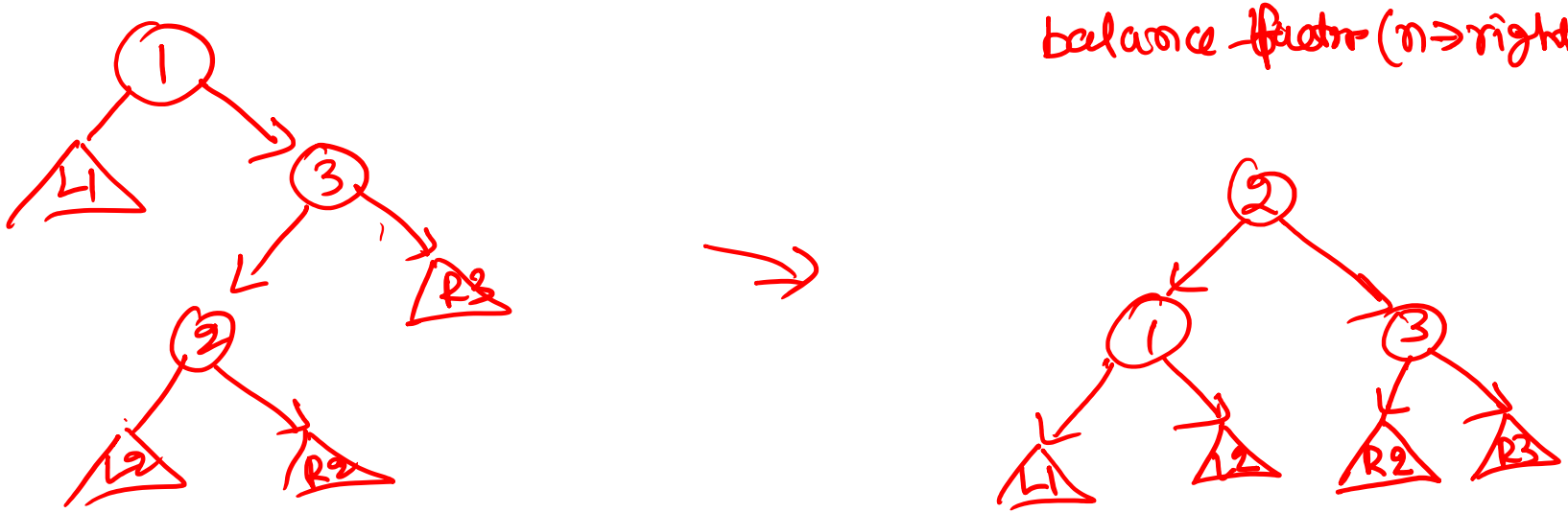
LR rotation

- We perform LR rotation on a node n, if
  - **balance_factor(n) > 1** and **n->left is right heavy**

```
Left heavy : balance factor > 0
Right heavy : balance factor < 0
```

RL rotation

if balance factor(n) < −1 and
balance factor(n→right) > 0



Left heavy : balance factor > 0
Right heavy : balance factor < 0

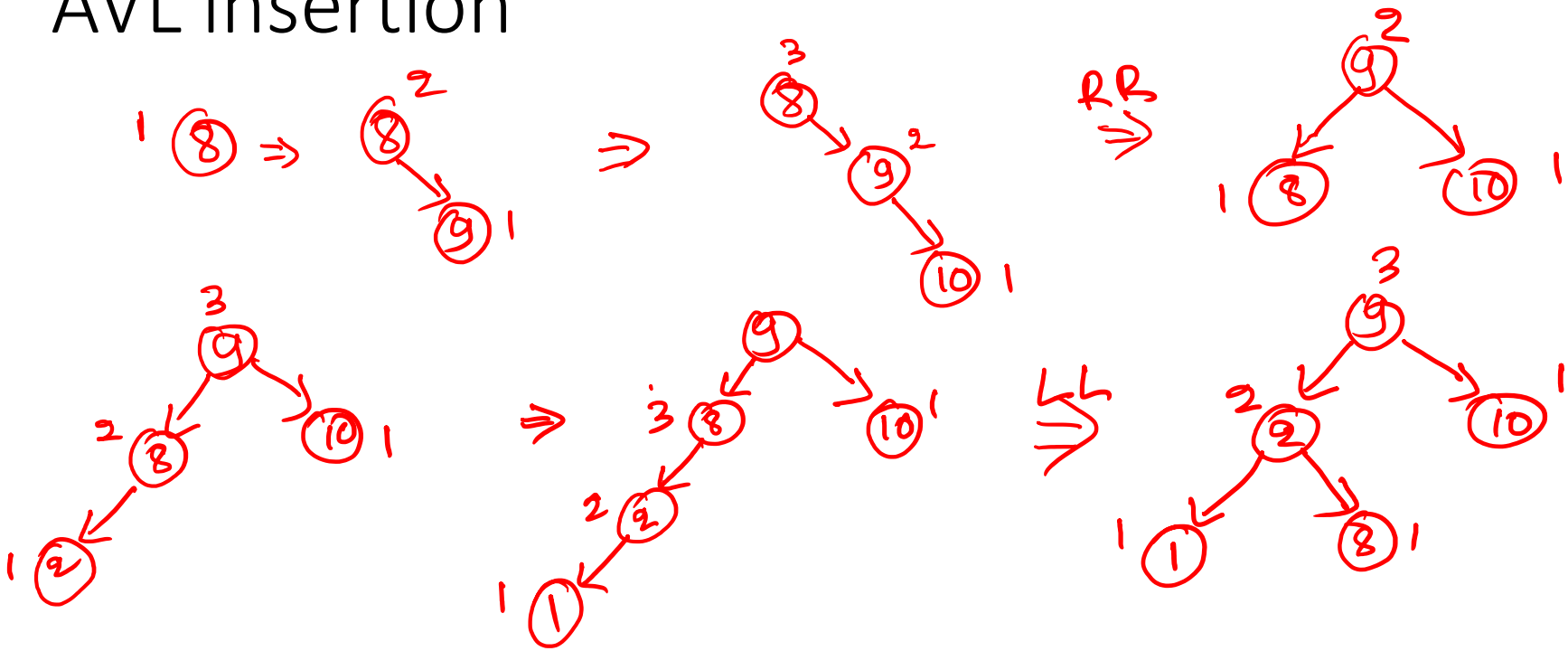
RL rotation

- We perform RL rotation on a node n, if
  - **balance_factor(n) < -1** and **n->right is left heavy**

```
Left heavy : balance factor > 0
Right heavy : balance factor < 0
```

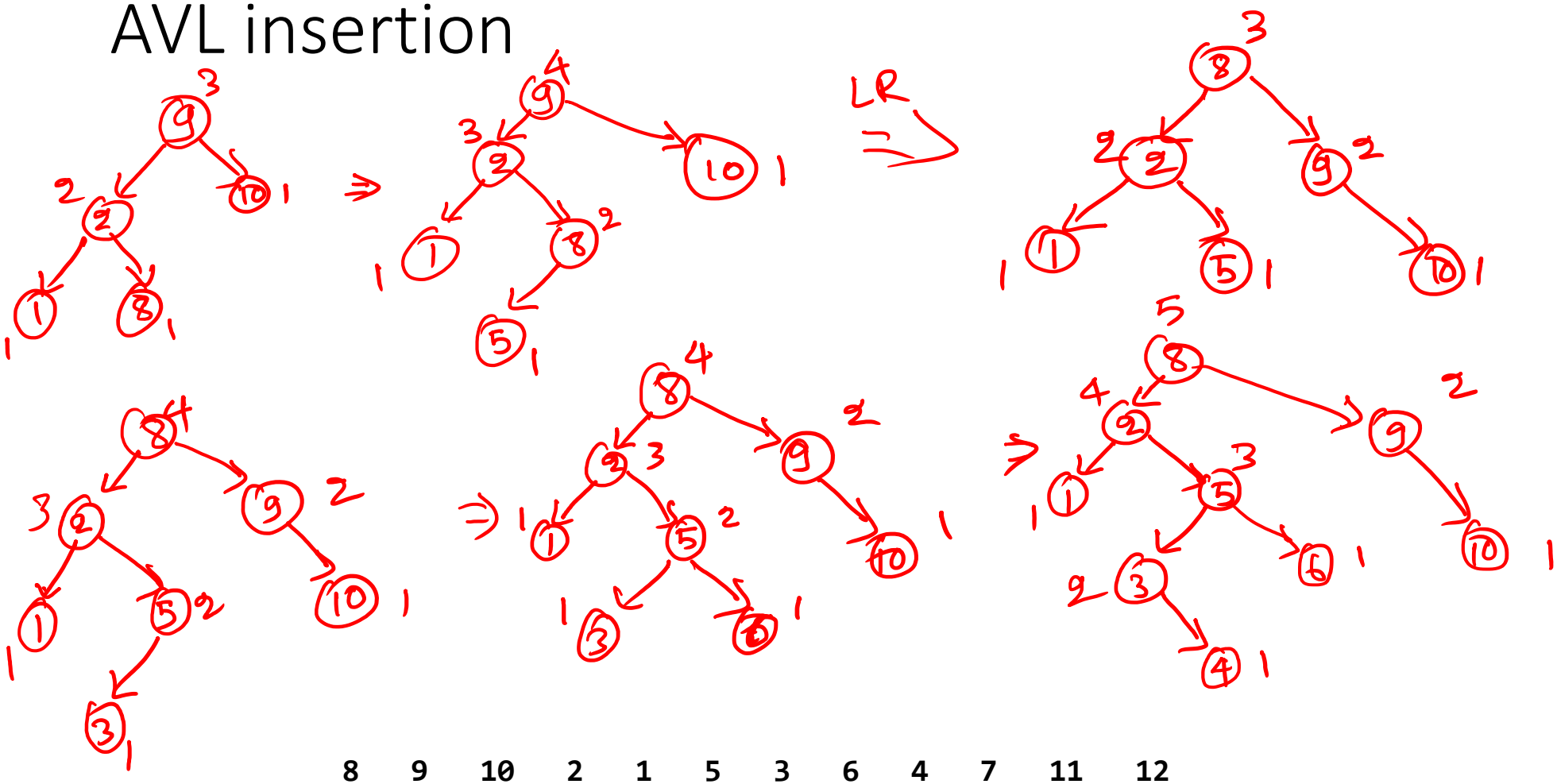
Insertion

AVL insertion



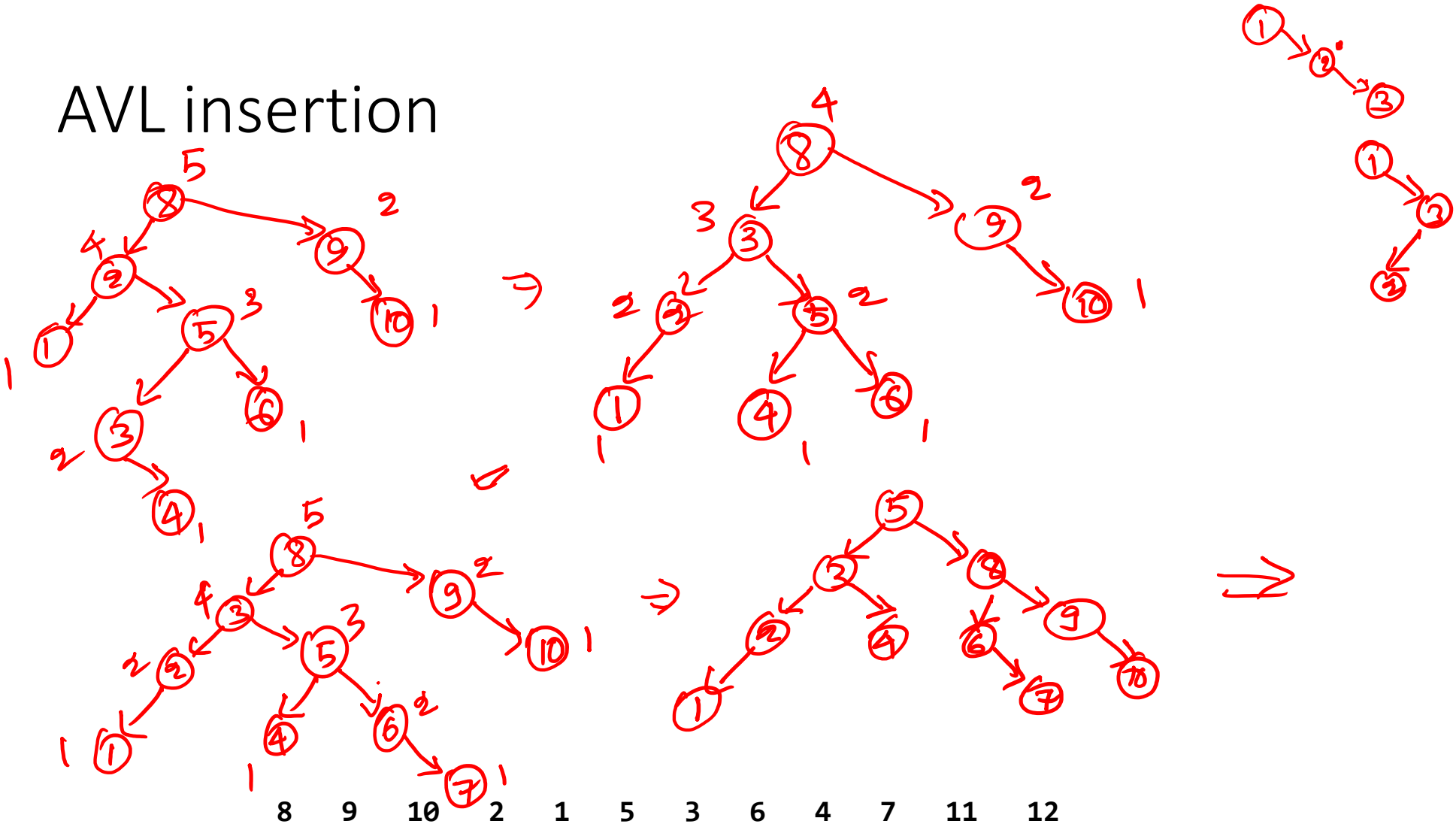
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



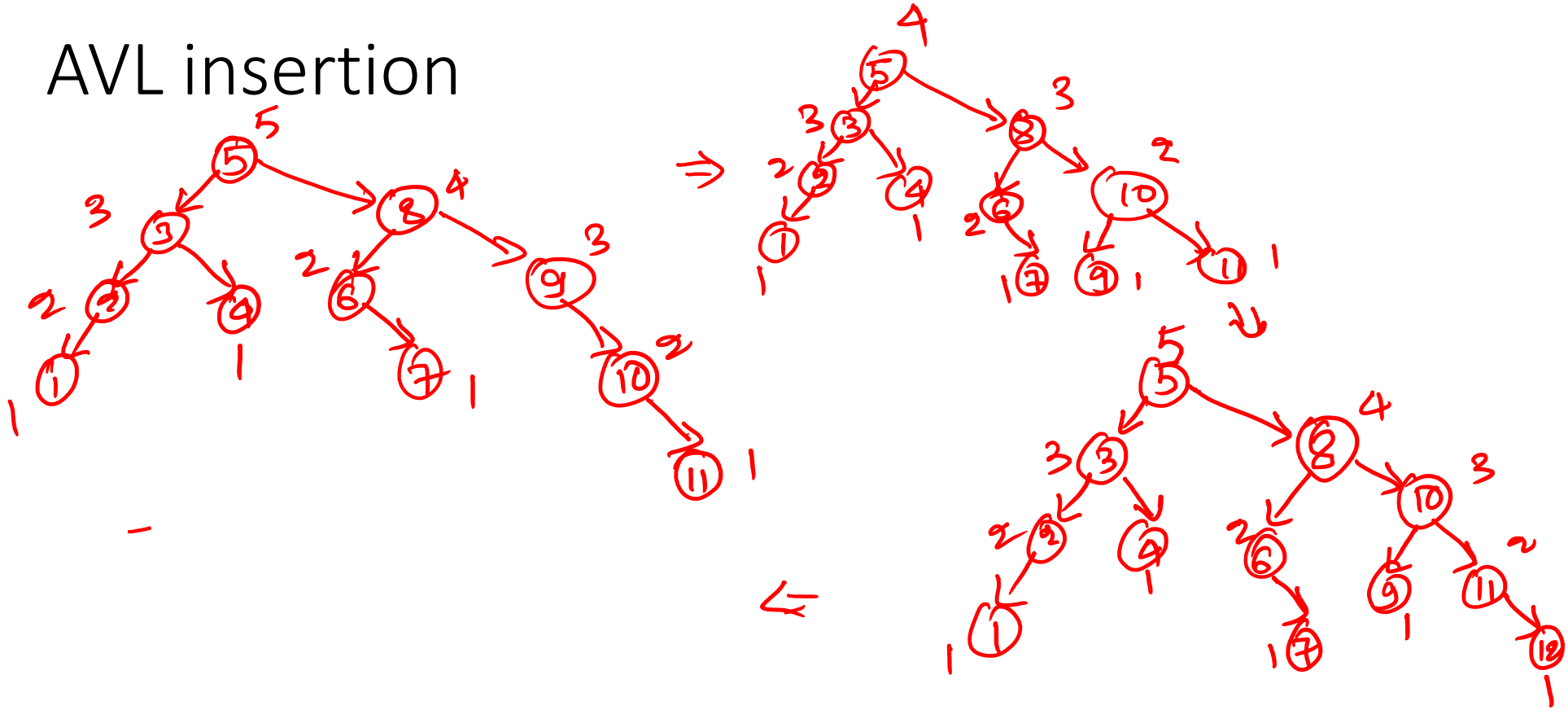
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



8 9 10 2 1 5 3 6 4 7 11 12

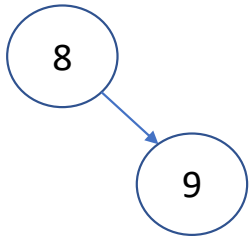
AVL insertion

8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion

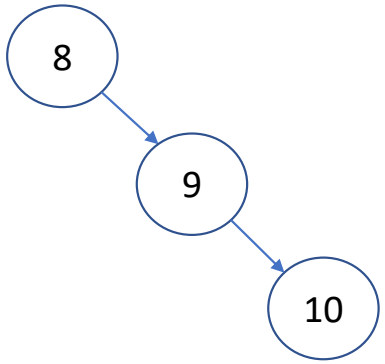
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



8 9 10 2 1 5 3 6 4 7 11 12

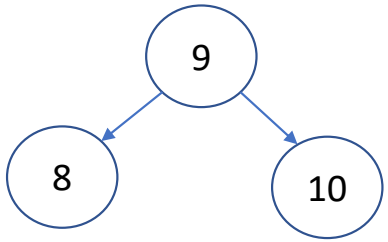
AVL insertion



need RR rotation

8 9 10 2 1 5 3 6 4 7 11 12

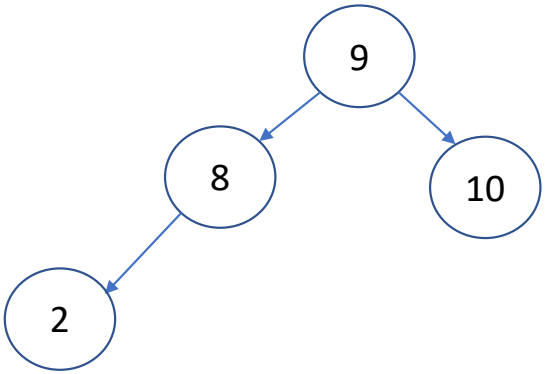
AVL insertion



after RR rotation

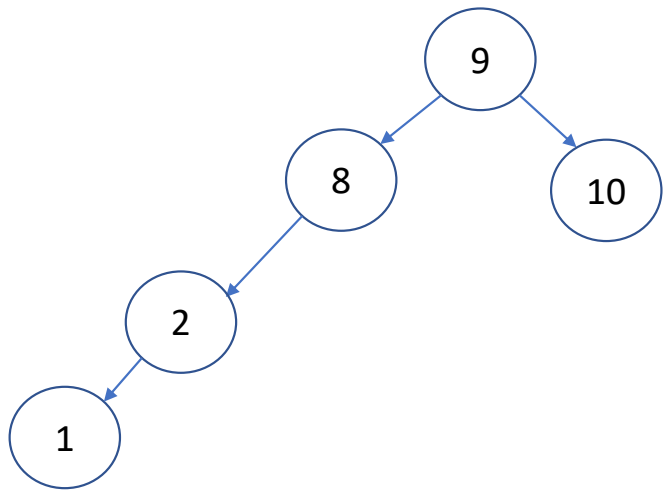
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



8 9 10 2 1 5 3 6 4 7 11 12

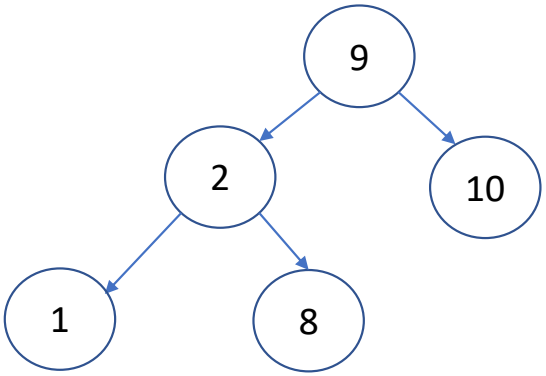
AVL insertion



need LL rotation

8 9 10 2 1 5 3 6 4 7 11 12

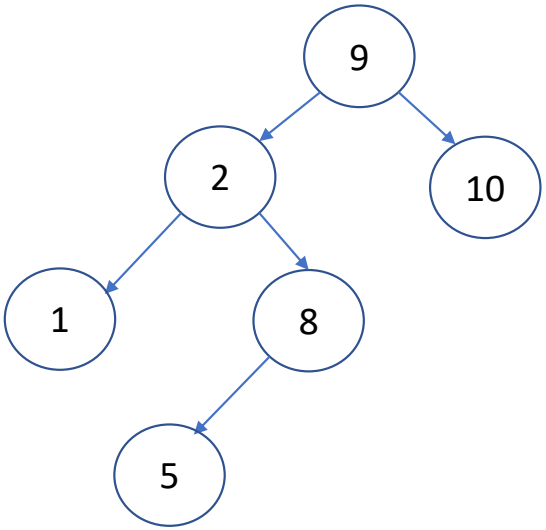
AVL insertion



after LL rotation

8 9 10 2 1 5 3 6 4 7 11 12

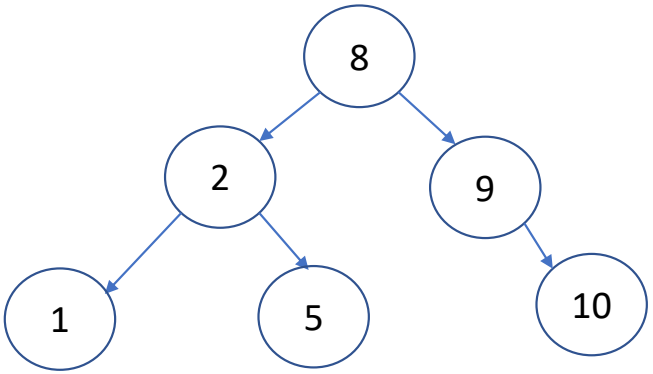
AVL insertion



need LR rotation

8 9 10 2 1 5 3 6 4 7 11 12

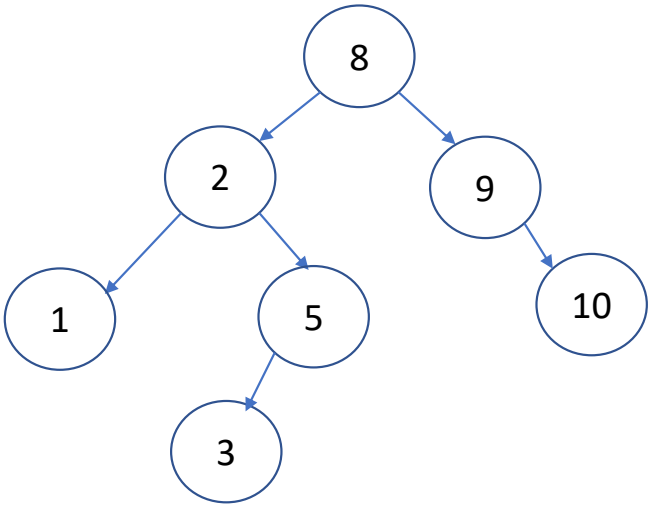
AVL insertion



after LR rotation

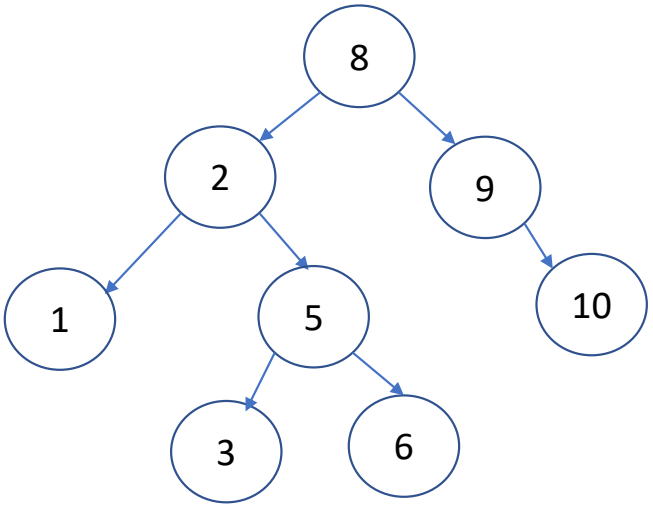
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



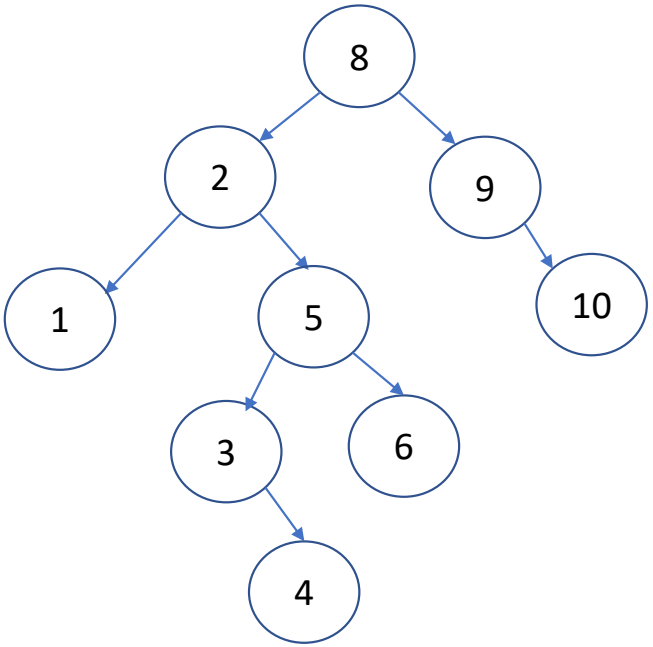
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



8 9 10 2 1 5 3 6 4 7 11 12

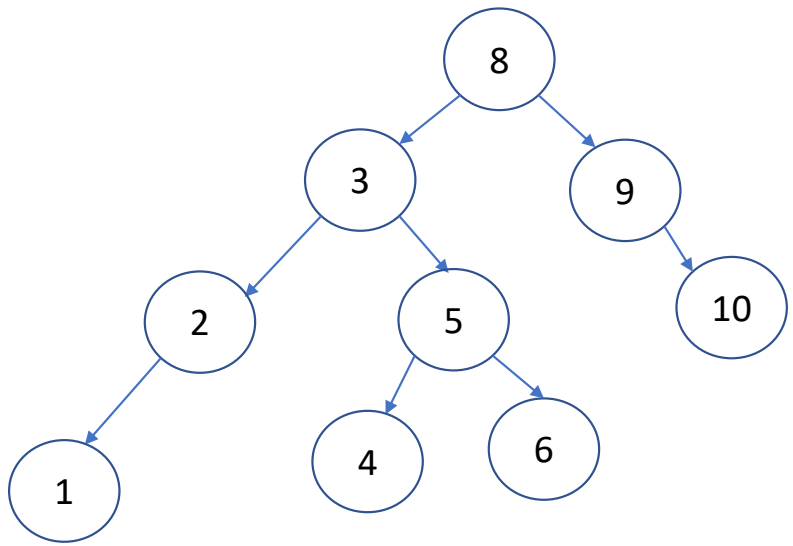
AVL insertion



need RL rotation

8 9 10 2 1 5 3 6 4 7 11 12

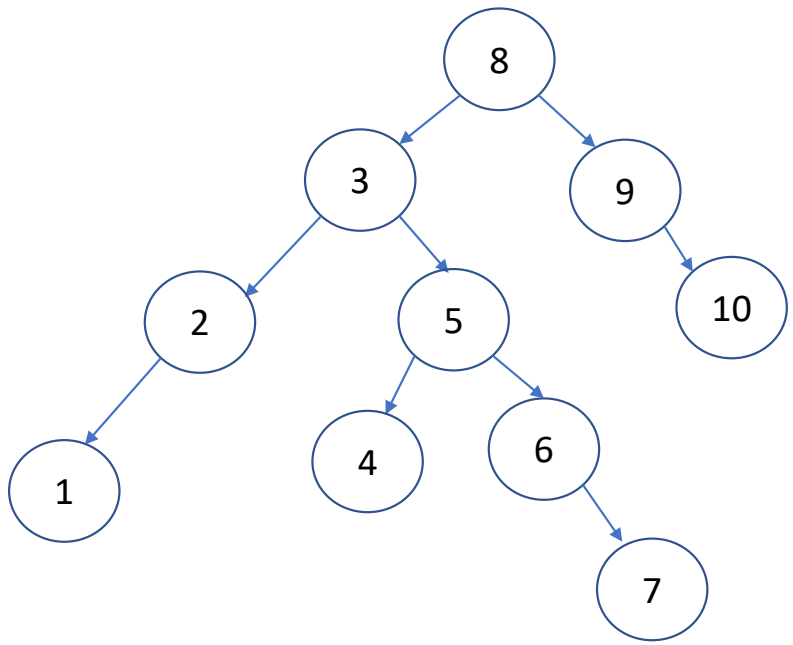
AVL insertion



after RL rotation

8 9 10 2 1 5 3 6 4 7 11 12

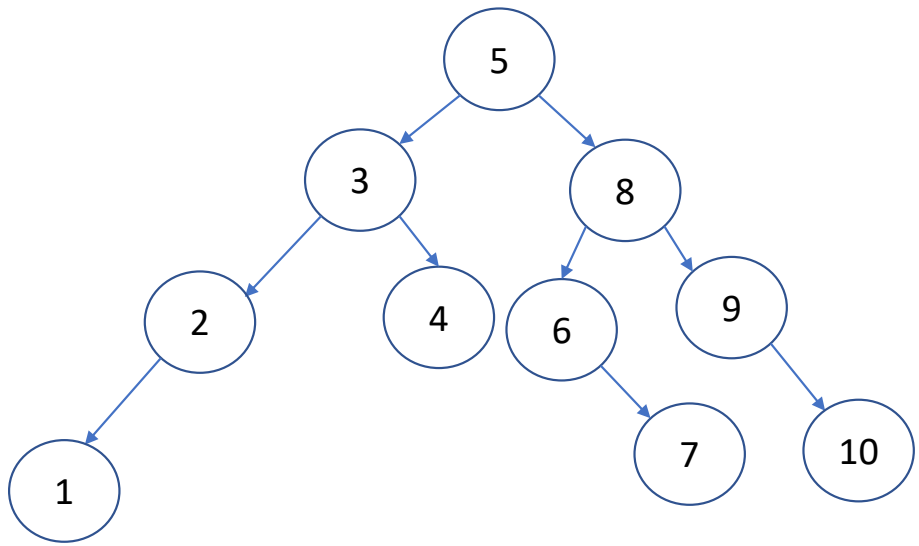
AVL insertion



need LR rotation

8 9 10 2 1 5 3 6 4 7 11 12

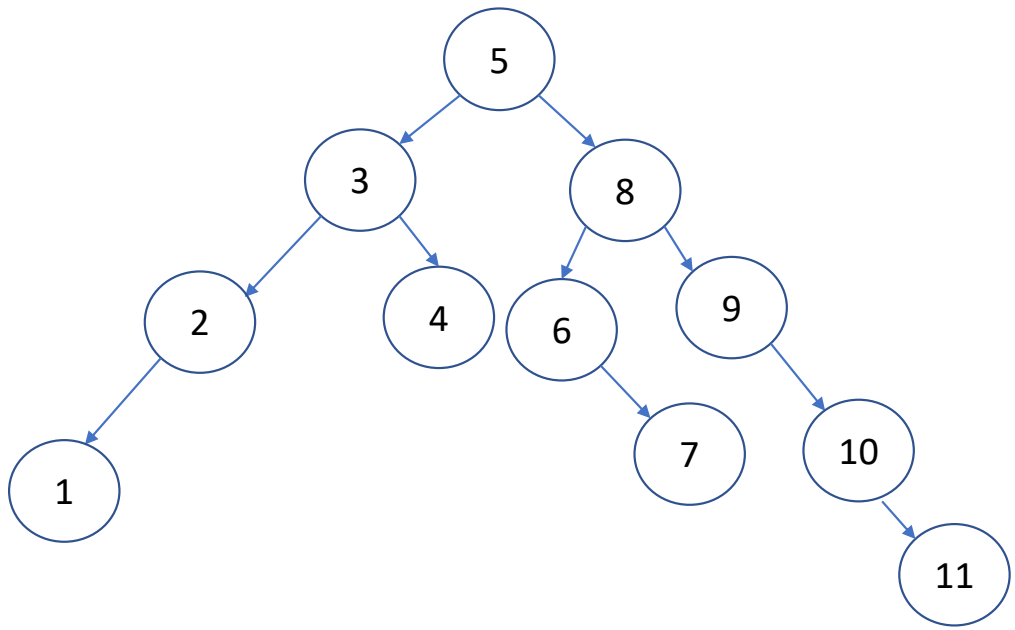
AVL insertion



after LR rotation

8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion

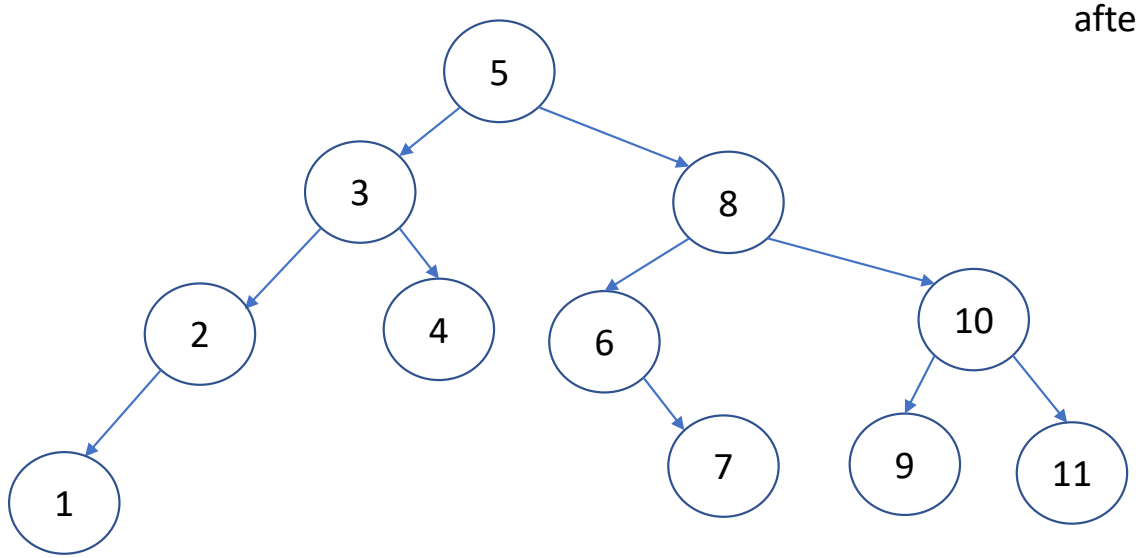


need RR rotation

8 9 10 2 1 5 3 6 4 7 11 12

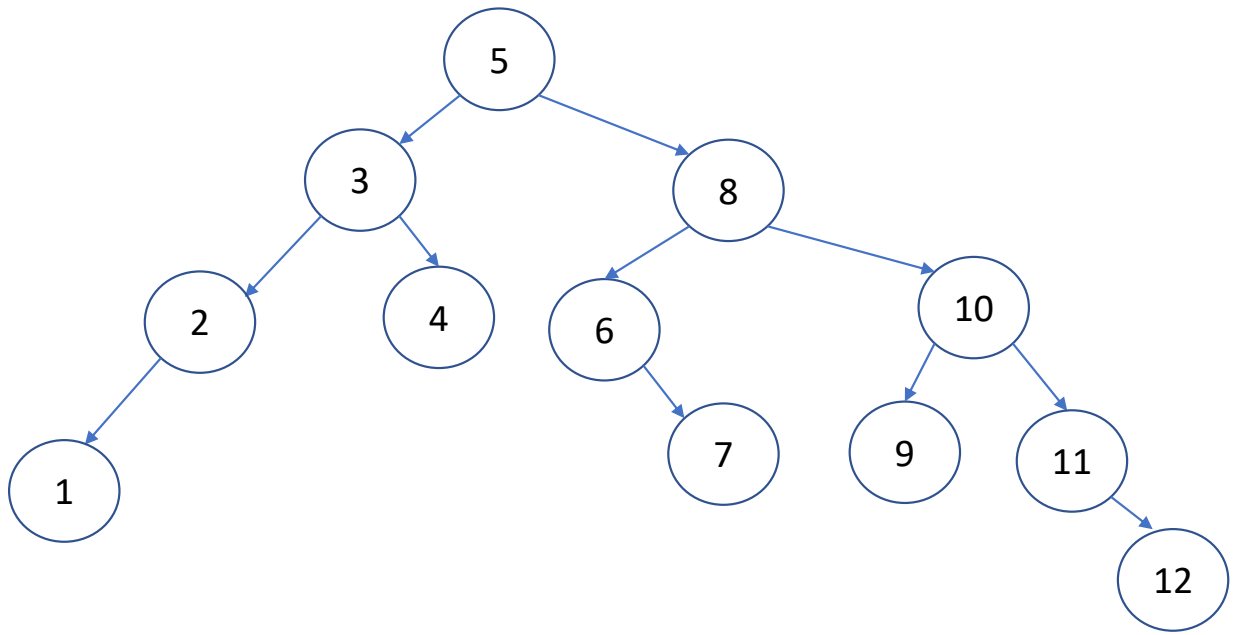
AVL insertion

after RR rotation



8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



8 9 10 2 1 5 3 6 4 7 11 12