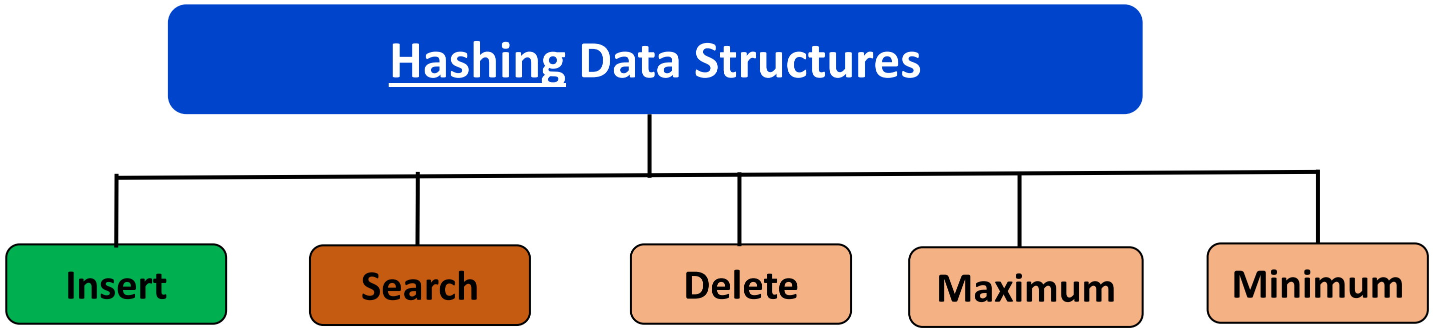
CS2x1:Data Structures and Algorithms

Koteswararao Kondepu

k.kondepu@iitdh.ac.in

Hashing (1)

Tables



Elements of Hashing:

i. *Hash Table* → *contains the key values with pointers to the corresponding records*
ii. *Hash Function*
iii. *Collisions*
iv. *Collision Resolution Techniques*

Hash Function

A "good" hash function minimizes the probability of collisions

Hash Functions:

- *Division method*
- *Midsquare method*
- *Folding method*
- *Multiplication method*

H(k) = k mod m ; k → key, m → table size; if index start from 0

H(k) = k (mod m) + 1; if index start from 1

Collision Resolution Technique:

- *Open addressing/Closed hashing*
  - *Linear probing method*
  - *Quadratic probing method*
  - *Double hashing method*
- *Closed addressing/Open hashing*
  - *Chaining*

H: K → I

K	I
10	1
19	0
35	8
43	7
62	8
59	4
31	4
49	3
77	4
33	6

Hash Function

0	19
1	10
2	
3	49
4	59, 31, 77
5	
6	33
7	43
8	35, 62
9	

Hash Table

Exercise: Division method

A **hash table of size 11** using the hash function **h(x) = x mod 11** . The key values are given in the following order: 44, 45, 46, 47, 33, and 55.

When we use Linear Probing for collision resolution, what is the index value of ***key 55****, if the index value starts from 0 ?*

h'(k) = k mod m ; k → key, m → table size;

H(k, i) = (h' (k) + i) mod m; i → {0, 1, 2, ...m -1}

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Exercise: Division method (1)

A **hash table of size 11** using the hash function **h(x) = x mod 11** . The key values are given in the following order: 41, 27, 9, 21, 22, 23, 34, and 45.

*When we use Quadratic Probing for collision resolution, what is the index value of **key 45**, if the index value starts from 0 ?*

h'(k) = k mod m ; k → key, m → table size;

H(k, i) = (h' (k) + i^2) mod m; i → {0, 1, 2, ….m -1}

0	22
1	23
2	34
3	
4	
5	
6	
7	
8	41
9	9
10	21

Exercise: Division method (2)

A **hash table of size 11** using the hash function **h(x) = x mod 11** . The key values are given in the following order: 41, 27, 9, 21, 13, 22, 23, 26, 2, and 30.

When we use Double hashing for collision resolution where:

$h_1(x) = x\ mod\ 11$

$h_2(x) = x \bmod 11$

what is the index value of ***key 30****, if the index value starts from 0 ?*

$h_1(k)$ = k mod m ; k → key, m → table size;

$h_2(k)$ = k mod m';

*h (k) = (h_1 (k) + i * h2(k)) mod m; i → {0, 1, 2, ….m -1}*

0	22
1	23
2	13
3	2
4	26
5	27
6	
7	
8	41
9	9
10	21

Hash Function: Generalization

$h_1(k) = k \bmod m$; $k \rightarrow$ key, $m \rightarrow$ table size;
$h_2(k) = k \bmod m'$;

$h(k) = (h_1(k) + F(i)) \bmod m$; $i \rightarrow \{0, 1, 2, \ldots.m - 1\}$

Hash Table

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Function: Midsquare

H(k) = x;
where x is obtained by selecting the appropriate number of bits or digits from
the middle of the square of the key value k

Keys (k):	*1234*	*2345*	*3456*
k^2:	$1\underline{5}2\underline{2}7\underline{5}6$	$5\underline{4}9\underline{9}0\underline{2}5$	$11\underline{9}4\underline{3}9\underline{3}6$

Policy (selection criteria): select 3 digits at even positions from the right most digit in the square

H(k):	*525*	*492*	*933*

Keys (k):	*1234*	*2345*	*3456*
k^2:	$152\underline{2}756$	$549\underline{9}025$	$1194\underline{3}936$

Policy (selection criteria): select middle digit (r) bits or digits → *range : 0 to* $2^r - 1$

H(k):	*2*	*9*	*3*

Exercise: Midsquare

Keys 101,121,131,141,151 are inserted into a hash Table of size 10 (0–9) using the <u>midsquare hash function</u> and <u>linear probing</u> is used for collision resolution. ***Use the middle digit of the square of the key value****. What is the index into which 151 will be inserted?*

A. 2
B. 8
C. 9
D. 6

Keys (k):	*101*	*121*	*131*	*141*	*151*
k^2:	*10<u>2</u>01*	*14<u>6</u>41*	*17<u>1</u>61*	*19<u>8</u>81*	*22<u>8</u>01*

Policy (selection criteria): select middle digit (r) bits or digits → range : 0 to 2^r - 1

H(k):	*2*	*6*	*1*	*8*	*8*

H(k) = x + i;

i → {0, 1, 2, ….m -1}

H(151, i=0) = (8+0) → ***Collision***
H(151, i=1) = (8+1)

Hash Table

0	
1	131
2	101
3	
4	
5	
6	121
7	
8	141
9	

Hash Function: Folding

$H(k) = k_1 + k_2 + k_3 + \ldots. + k_n$;

When the key is a large number, key is partition into multiple parts such as k_1 k_2 k_3 … k_n
Note: if the keys are in binary form, the exclusive-OR operation may be substituted in place of addition

Keys (k):	*1522756*	*5499025*	*11943936*
Chopping:	*01 52 27 56*	*05 49 90 25*	*11 94 39 36*
Pure folding:	*01 + 52 + 27 + 56 = 136*	*05 + 49 + 90 +25 = 169*	*11 + 94 + 39 + 36 = 180*
Fold shifting: $[k_1, k_3, k_5 \ldots]$	*10 + 52 + 72 + 56 = 190*	*50 + 49 + 09 + 25 = 133*	*11 + 94 + 93 + 36 = 234*
Fold boundary: $[k_1, k_n]$	*10 + 52 + 27 + 65 = 154*	*50 + 49 + 90 + 52 = 241*	*11 + 94 + 39 + 63 = 207*

Exercise: Folding

Consider the following keys are inserted into a hash Table of size 10 (0−9) using the *folding hash function* and *Linear Probing is used for collision resolution*.

Keys (k): 4765, 7052, 6745, 6574, 7502

What is the index into which 6574 will be inserted?

Keys (k):	*4764*	*7052*	*6745*	*6574*	*7502*
Chopping:	*47 64*	*70 52*	*67 45*	*65 74*	*75 02*
Pure folding:	*47 + 64 = 111*	*70 + 52 = 122*	*67 + 45 = 112*		*75 + 02 = 79*

$H(k) = k_1 + k_2 + k_3 + \ldots + k_n;$

When the key is a large number, key is partition into multiple parts such as $k_1\ k_2\ k_3 \ldots k_n$
Note: if the keys are in binary form, the exclusive-OR operation may be substituted in place of addition

Hash Function: Multiplication

$H(k) = \lfloor m * (kA \bmod 1) \rfloor$; $k \rightarrow$ key, $0 < A < 1$

$kA \bmod 1 \rightarrow$ fractional

$A = 0.1952$; Key $(k) = 5$; $m = 5$

$kA = 0.976$

$$
\begin{aligned}
H(5) &= \lfloor m * (kA \bmod 1) \rfloor \\
&= \lfloor 5 * (0.976 \bmod 1) \rfloor \\
&= \lfloor 5 * 0.976 \rfloor \\
&= \lfloor 4.88 \rfloor \\
&= 4
\end{aligned}
$$

$$
\begin{aligned}
H(10) &= \lfloor m * (kA \bmod 1) \rfloor \\
&= \lfloor 5 * (1.952 \bmod 1) \rfloor \\
&= \lfloor 5 * 0.952 \rfloor \\
&= \lfloor 4.76 \rfloor \\
&= 4 \rightarrow \textbf{Collision}
\end{aligned}
$$

→ *apply linear probing or any other collision resolution techniques*

Drawbacks: Open addressing → Closed hashing

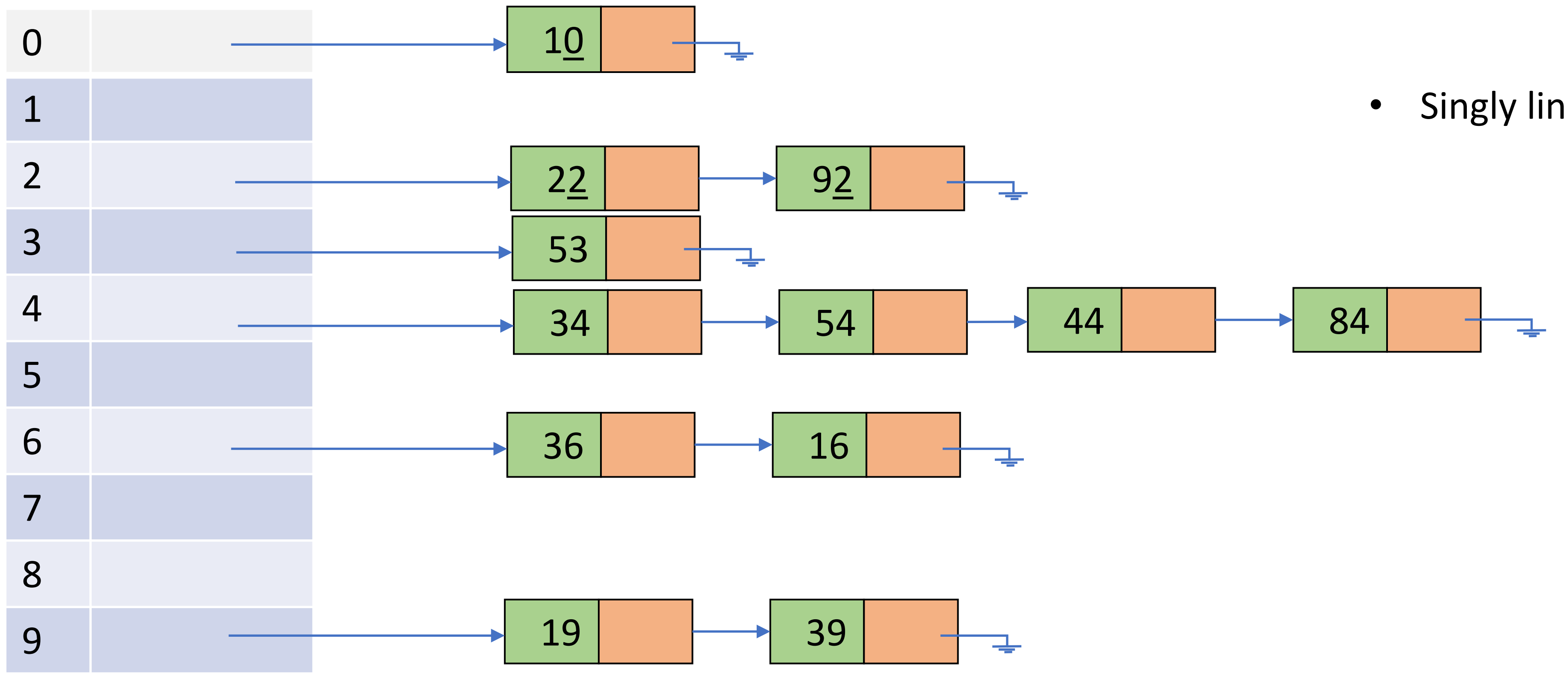
- *All the key values are stored in the hash tables*
- *The closed hashing mostly deals with array-based implementation for hash tables*
  - *Difficult to handle the situation of hash table overflows*
  - *The majority of the keys far from their hash location → increasing the number of probes → decreases the overall performance*

To resolve the closed hashing problem → Open hashing (separate chaining or simply chaining)

Open hashing

- *Open hashing is also called as separate chaining or simply chaining*
- *Chaining method uses a hash table as an array of pointers → each pointer points to a linked list*

Table size = 10; Policy: H(k) → the last digit of the key values



- Singly linked list

Advantages and disadvantages of chaining

- An overflow scenario never arises. The hash table is not restricted with the size, hence it can hold any number of key values
- Collision resolution can be achieved very efficiently, if the list maintained an ordering of keys → keys can be searched quickly
- Insertion and deletion becomes a quick and easy task in open hashing. Deletion of a key follows the same way of deleting a node in a singly linked list
- Open hashing is best suitable in applications where the number of keys values varies drastically → due to dynamic storage management policy
- *Open hashing required an additional storage space for maintaining linked lists and its link fields.*

Comparison of Collision Resolution Techniques

- Analytical comparison of various collision resolution techniques is measured by using <u>*load factor*</u>

$$load\ factor\ \alpha = \frac{Total\ number\ of\ key\ values}{Size\ of\ the\ hash\ table} = \frac{n}{m}$$

Arrays: Operations

Index:	a[0]	a[1]	a[2]	a[3]				a[n-1]
	10	**20**	**30**	**40**	**…….**	**……**	**….**	**101**

- Insertion
  - ✓ Best case : <u>*Insertion at the end*</u> → $\Omega(1)$
  - ✓ Worst case : <u>*Insertion at the beginning*</u> → $O(n)$
  - ✓ Average case : <u>*Insertion in middle*</u> → $\theta(n)$

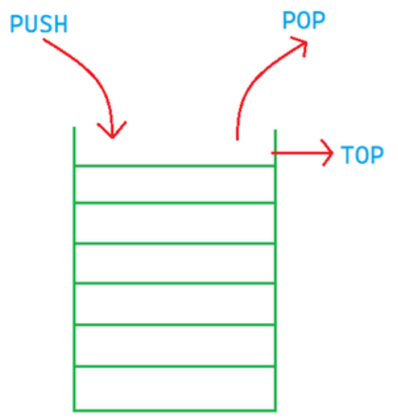
- Deletion
  - ✓ Best case : <u>*Deletion at the end*</u> → $\Omega(1)$
  - ✓ Worst case : <u>*Deletion at the beginning*</u> → $O(n)$
  - ✓ Average case : <u>*Deletion in middle takes*</u> $\theta(n)$

- Traversal/lookup
  - ✓ Direct access : O (1) [can access any element with base address]

Stack and Queue: Operations

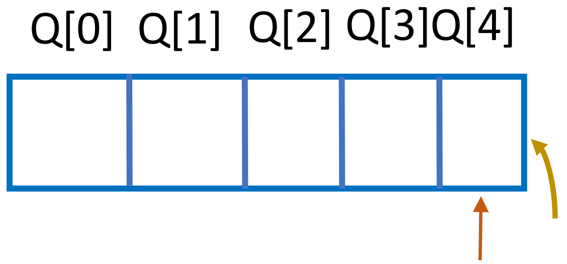
❖ Stack Operations

- ✓ Push : *Insertion at the top* → $O(1)$
- ✓ Pop : *Delete from the top* → $O(1)$
- ✓ IsEmpty : *Exception handling* → $O(1)$
- ✓ IsFull : *Exception handling* → $O(1)$

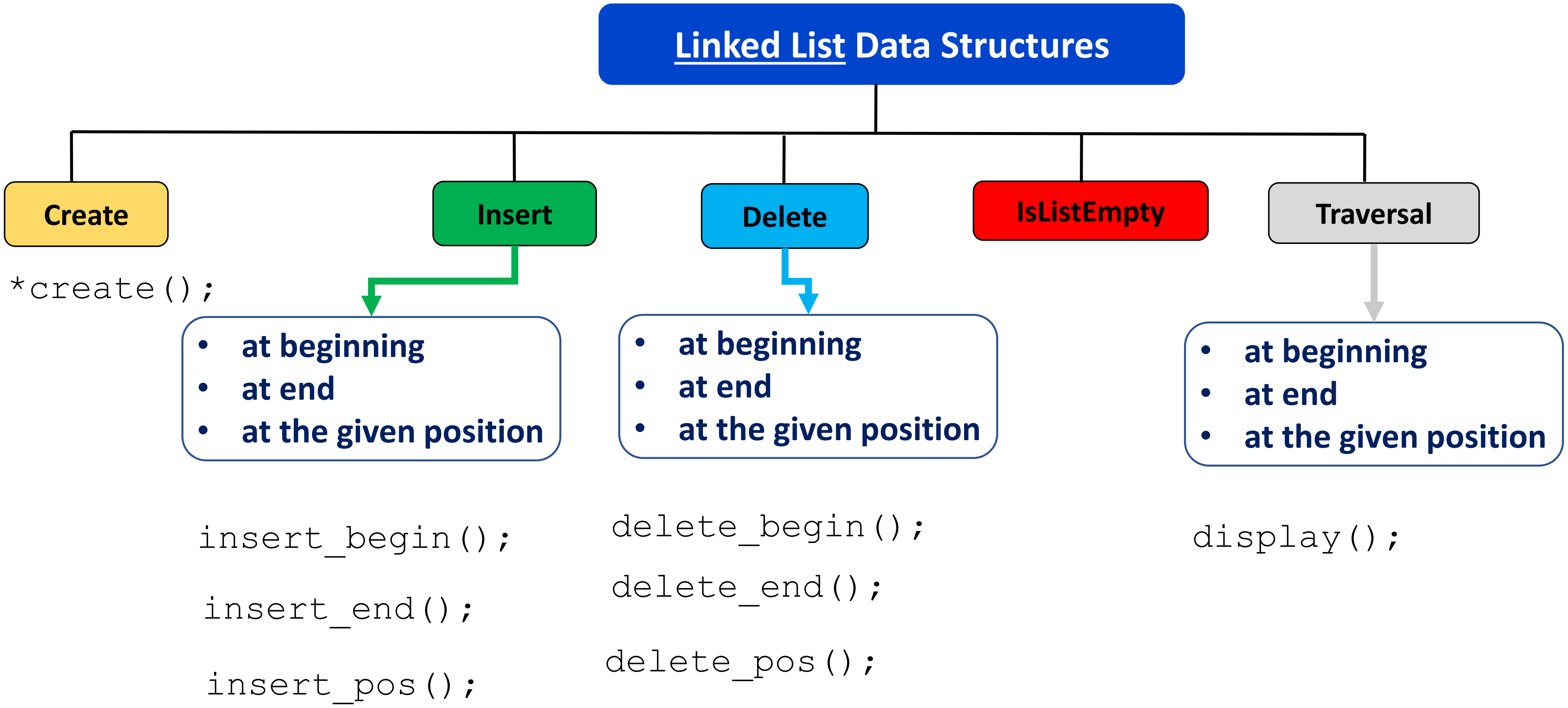


❖ Queue Operations

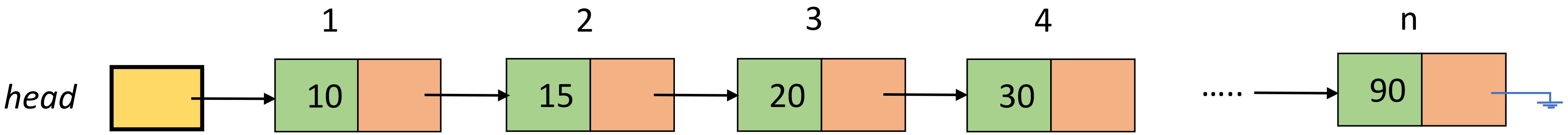
- ✓ EnQueue : *Insertion at the tail* → $O(1)$
- ✓ DeQueue : *Delete from the front* → $O(1)$
- ✓ IsEmpty : *Exception handling* → $O(1)$
- ✓ IsFull : *Exception handling* → $O(1)$



Linked List: Operations



Singly Linked List: Traversal

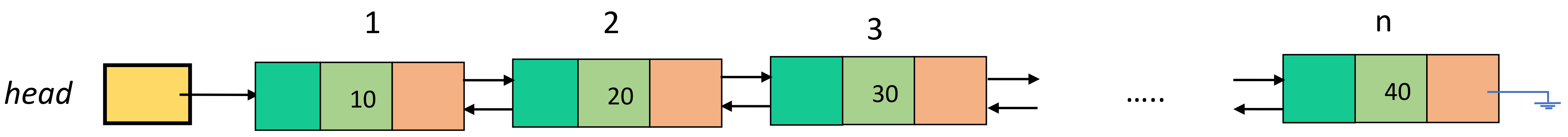


traversal

1. *struct node *traversal*
2. *traversal = head;*
3. *while (traversal != NULL)*
   display the element: traversal → data
   traversal = traversal →next

Time Complexity O (n) → for visiting the list of size n
Space Complexity O (1) → for creating a temporary variable → traversal

Doubly Linked List: Traversal



traversal

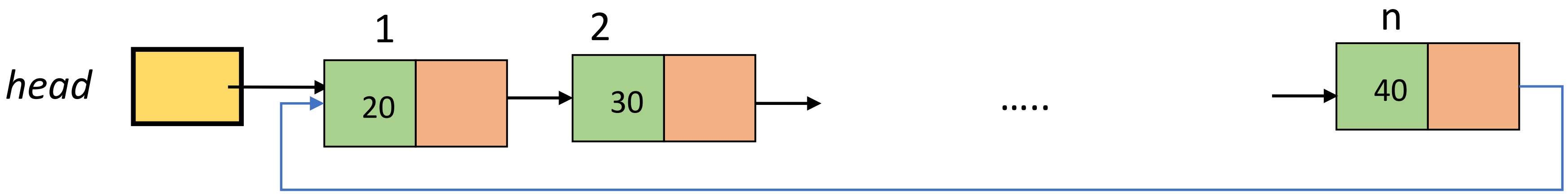
1 *struct node *traversal*

2 *traversal = head;*

3 *while (traversal != NULL)*
display the element: traversal → data
traversal = traversal →next

Time Complexity O (n) → for visiting the list of size n
Space Complexity O (1) → for creating a temporary variable → traversal

Circular Singly Linked List: Traversal

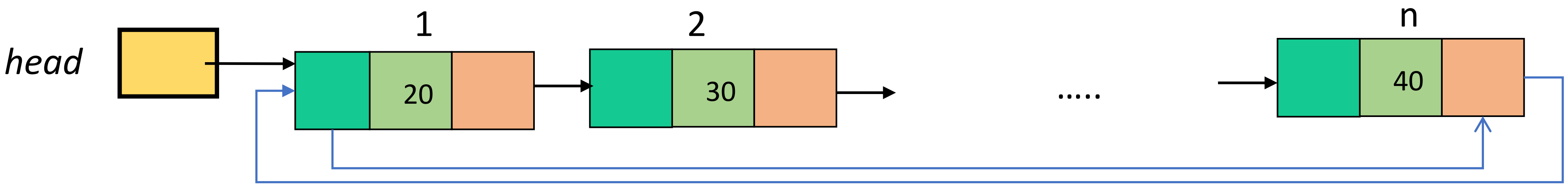


traversal

1. *struct node *traversal* *traversal* 
2. *traversal = head;*
3. *while (**traversal → next != head**)*
   display the element: traversal → data
   traversal = traversal →next
4. ***display the last element: traversal → data***

Time Complexity O (n) → for visiting the list of size n
Space Complexity O (1) → for creating a temporary variable → traversal

Circular Doubly Linked List: Traversal



traversal

1 struct node *traversal   traversal 

2 traversal = head;

3 while (**traversal → next != head**)
    display the element: traversal → data
    traversal = traversal →next

4 **display the last element: traversal → data**

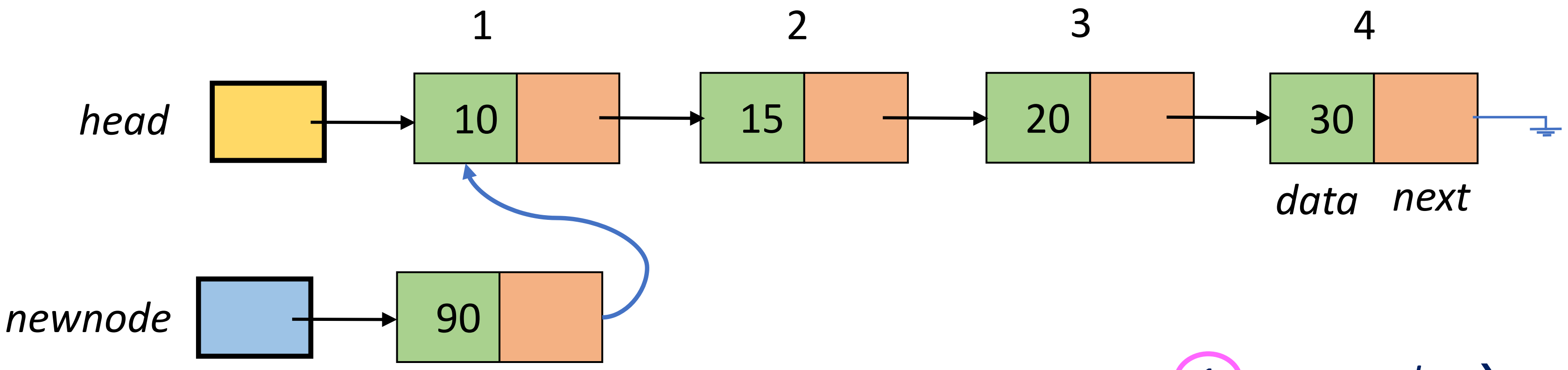
Time Complexity O (n) → for visiting the list of size n

Space Complexity O (1) → for creating a temporary variable → traversal

Linked List: Traversal

Linked Lists Traversal Operation	Time Complexity analysis
Singly Linked List	O (n)
Doubly Linked List	O (n)
Circular Singly Linked List	O (n)
Circular Doubly Linked List	O (n)

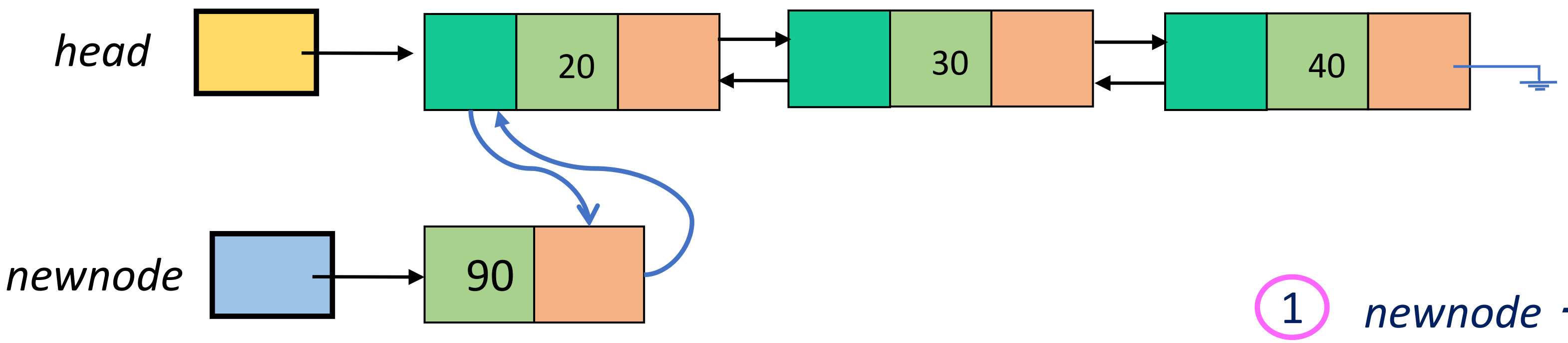
Singly Linked List: Insert at the beginning



1. *newnode → next = head*
2. *head = newnode*

Time Complexity O (1) → for modifying the above two steps
Space Complexity O (1) → for creating a newnode

Doubly Linked List: Insert at the beginning



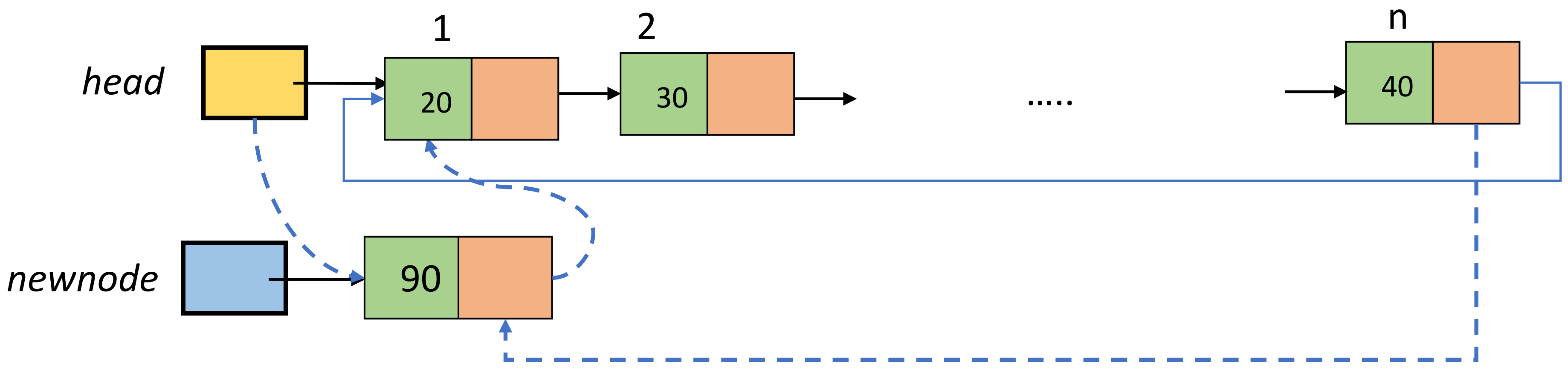
1 *newnode → next = head*

2 *head → prev = newnode*

3 *head = newnode*

Time Complexity O (1) → for modifying the above two steps
Space Complexity O (1) → for creating a newnode

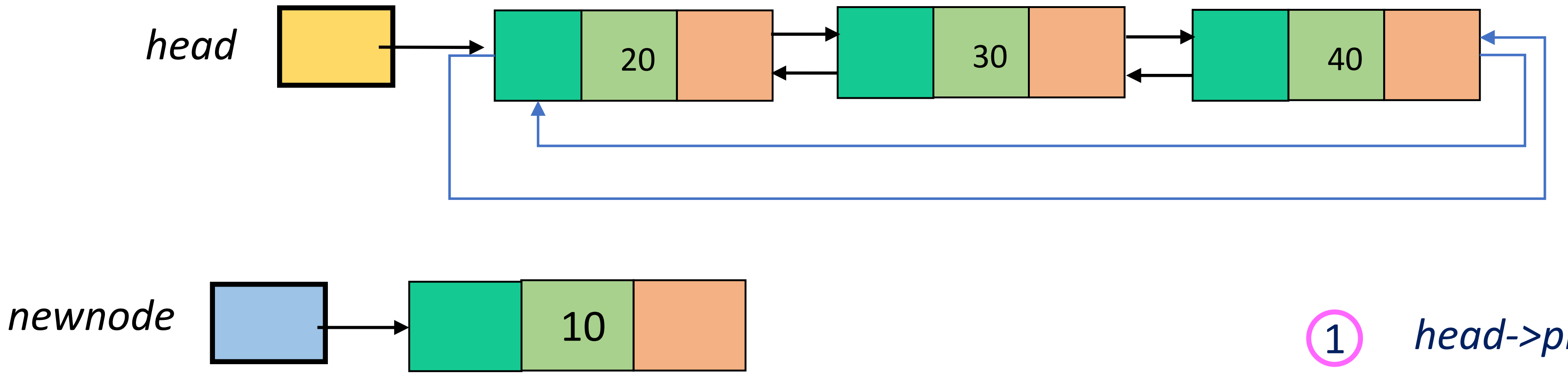
Circular Singly Linked List: Insert at the beginning



Time Complexity O (n) → for traversing the complete list of size n
Space Complexity O (1) → for creating a newnode

1. *begin = head*
   while (begin → next != head)
   begin = begin → next
2. *begin->next = newnode;*
3. *newnode->next=head;*
4. *head=newnode;*

Circular Doubly Linked List: Insert at the beginning

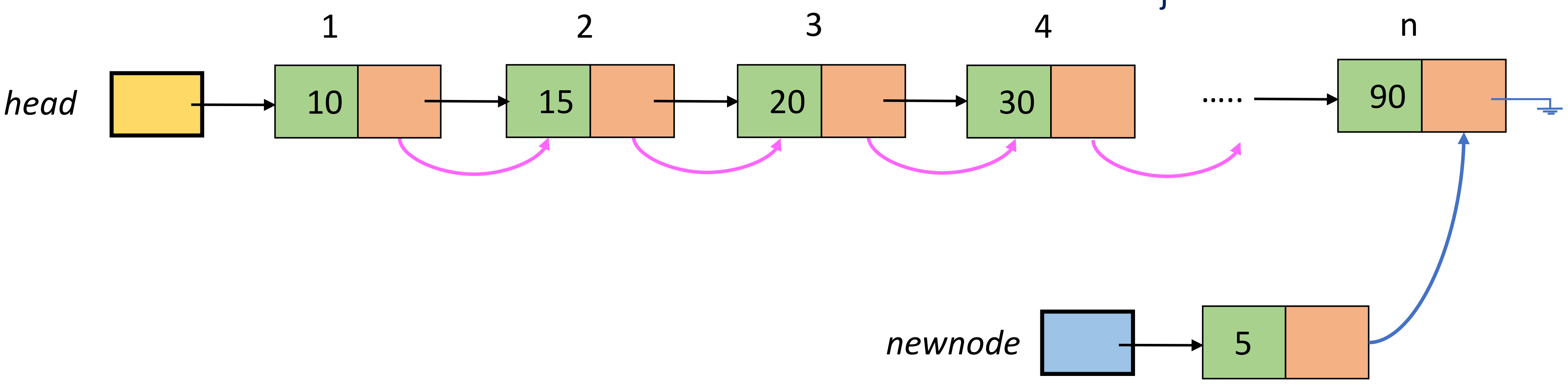


1. *head->prev->next = newnode;*
2. *newnode->prev=head->prev;*
3. *head->prev=newnode;*
4. *newnode->next=head;*
5. *head=newnode;*

Time Complexity O (1) → for modifying the five steps
Space Complexity O (1) → for creating a newnode

Singly Linked List: Insert at the end

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

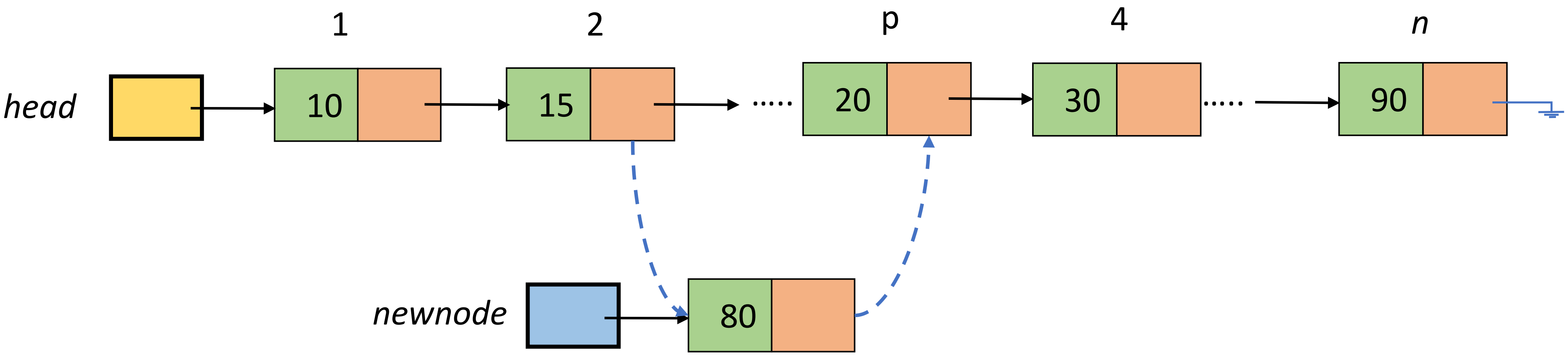


Time Complexity O (n) → for visiting the list of size n
Space Complexity O (1) → for creating a newnode

1 *while (tail → next != NULL)*
tail = tail → next

Singly Linked List: Insert at the given position

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



Time Complexity O (n) → In the worst case, it may happen that we need to insert at the end
Space Complexity O (1) → for creating a newnode

Linked List: Insertion

Linked Lists ***Insertion at the beginning*** Operation	Time Complexity analysis
Singly Linked List	O (1)
Doubly Linked List	O (1)
Circular Singly Linked List	O (n)
Circular Doubly Linked List	O (1)

Linked Lists ***Insertion at the end*** Operation	Time Complexity analysis
Singly Linked List	O (n)
Doubly Linked List	O (n)
Circular Singly Linked List	O (n)
Circular Doubly Linked List	O (1)

Linked List: Deletion

Linked Lists ***Deletion at the beginning*** Operation	Time Complexity analysis
Singly Linked List	O (1)
Doubly Linked List	O (1)
Circular Singly Linked List	O (n)
Circular Doubly Linked List	O (1)

Linked Lists ***Deletion at end*** Operation	Time Complexity analysis
Singly Linked List	O (n)
Doubly Linked List	O (n)
Circular Singly Linked List	O (n)
Circular Doubly Linked List	O (1)

Binary Tree

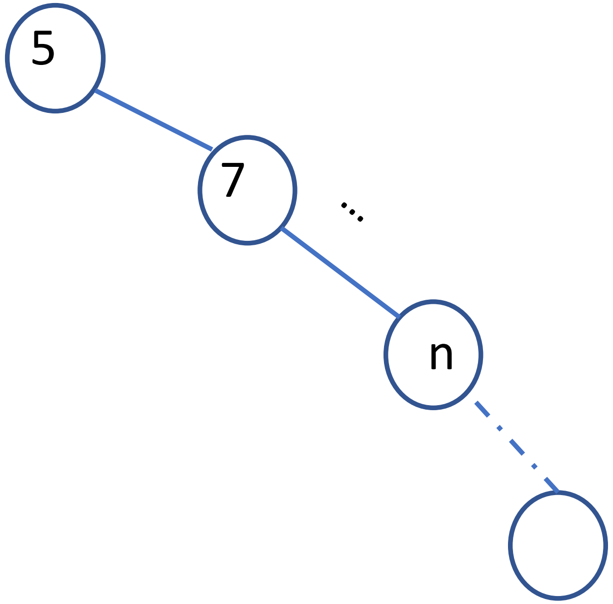
<u>Time complexity analysis</u>:

❖ Insertion
- ✓ Best case : *<u>Insertion at the root node</u>* → $\Omega(1)$
- ✓ Worst case : *<u>Insertion in any of Left/Right skew trees</u>* → $O(n)$
- ✓ Average case : *<u>Insertion in a balanced tree</u>* → $\theta(log(n))$

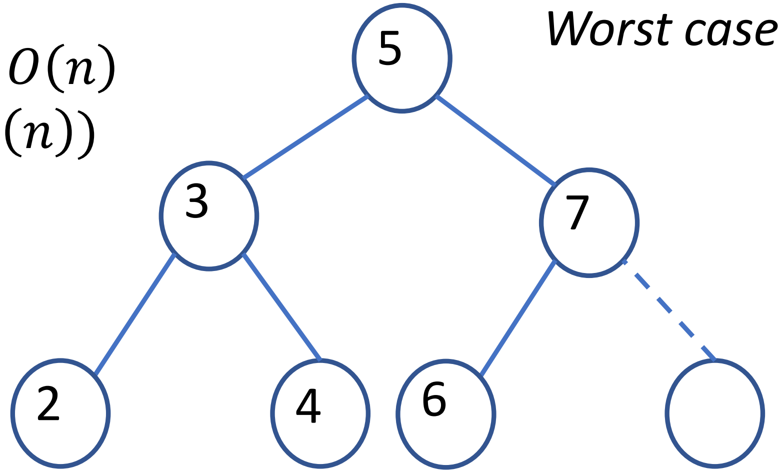
❖ Search
- ✓ Best case : Found at the root node → $\Omega(1)$
- ✓ Worst case : For any of the Left/Right skew tree takes → $O(n)$
- ✓ Average case : Searching in a balanced tree takes → $\theta(log(n))$

❖ Deletion
- ➢ Best case : Deletion at the root node takes → $\Omega(1)$
- ➢ Worst case : Leaf node of any Left/Right skew → $O(n)$
- ➢ Average case : Deletion of a node [except level 0] in a balanced tree → $\theta(log(n))$ time

Best case



Worst case



Average case

Binary tree

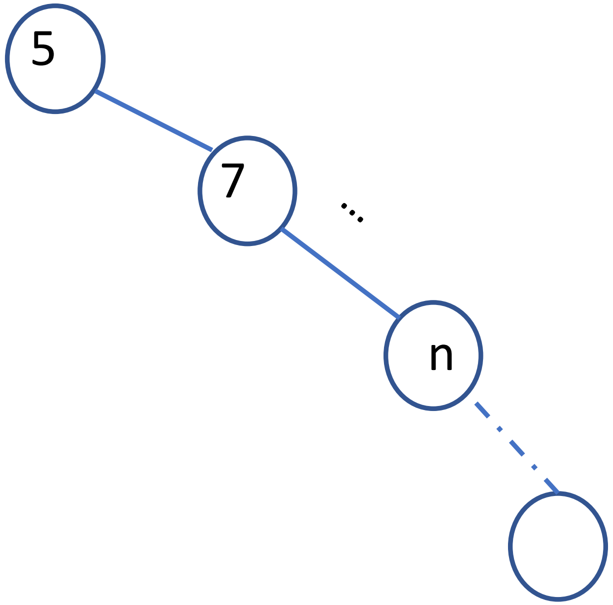
<u>Time complexity analysis:</u>

Operation	Best Case	Average Case	Worst Case
Insert	$\Omega(1)$	$\theta(log(n))$	$O(n)$
Search	$\Omega(1)$	$\theta(log(n))$	$O(n)$
Deletion	$\Omega(1)$	$\theta(log(n))$	$O(n)$

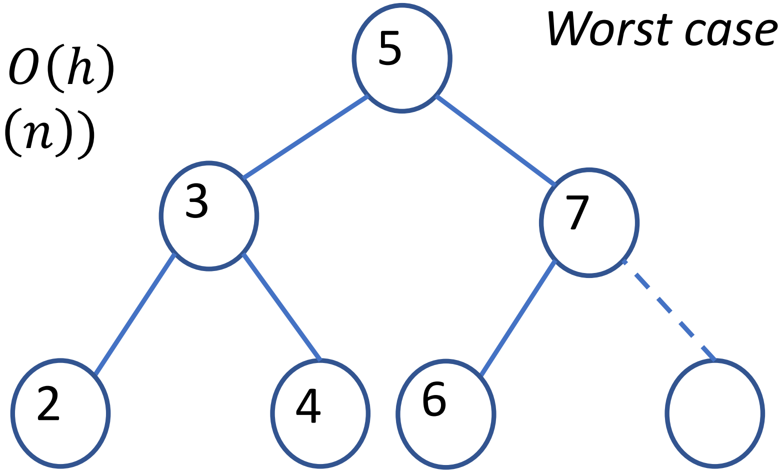
Binary Search Tree

__Time complexity analysis:__

Best case



Worst case



Average case

❖ Insertion
- ✓ Best case : *Insertion at the root node* → $\Omega(1)$
- ✓ Worst case : *Insertion in any of Left/Right skew trees* → $O(h)$
- ✓ Average case : *Insertion in a balanced tree* → $\theta(log(n))$

❖ Search
- ✓ Best case : Found at the root node → $\Omega(1)$
- ✓ Worst case : For any of the Left/Right skew tree takes → $O(h)$
- ✓ Average case : Searching in a balanced tree takes → $\theta(log(n))$

❖ Deletion
- ➢ Best case : Deletion at the root node takes → $\Omega(1)$
- ➢ Worst case : Leaf node of any Left/Right skew → $O(h)$
- ➢ Average case : Deletion of a node [except level 0] in a balanced tree → $\theta(log(n))$ time

Binary Search tree

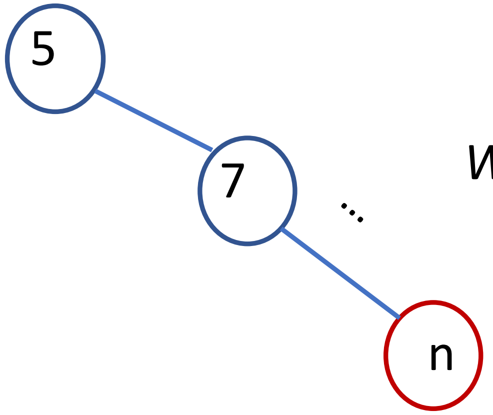
Time complexity analysis:

Operation	Best Case	Average Case	Worst Case
Insert	$\Omega(1)$	$\theta(log(n))$	$O(h)$
Search	$\Omega(1)$	$\theta(log(n))$	$O(h)$
Deletion	$\Omega(1)$	$\theta(log(n))$	$O(h)$

AVL Tree

<u>Time complexity analysis</u>:

Best case



Worst case

❖ Insertion

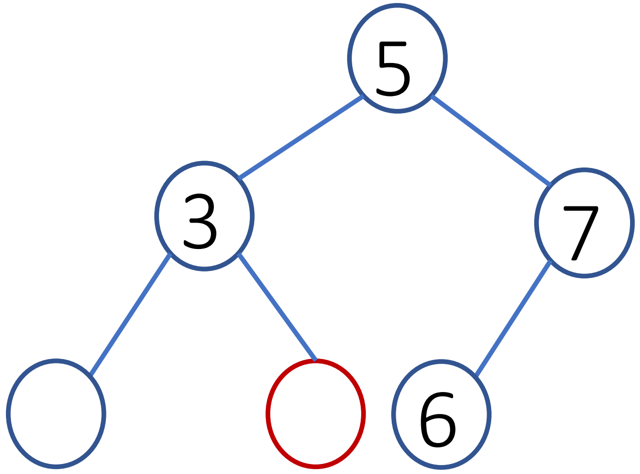
- ✓ Best case : *<u>Insertion at the root node</u>* → $\Omega(1)$
- ✓ Worst case : *<u>Insertion trees</u>* → $O(log(n))$
- ✓ Average case : *<u>Insertion in a balanced AVL tree</u>* → $\theta(log(n))$

❖ Search

- ✓ Best case : *<u>Found at the root node</u>* → $\Omega(1)$
- ✓ Worst case : *<u>The AVL tree takes</u>* → $O(log(n))$
- ✓ Average case : *<u>Searching in a balanced tree</u>* → $\theta(log(n))$

❖ Deletion

- ✓ Best case : *<u>Deletion at the root node</u>* → $\Omega(1)$
- ✓ Worst case : *<u>Leaf node of any AVL tree</u>* → $O(log(n))$
- ✓ Average case : *<u>Deletion of a node in an AVL tree</u>* → $\theta(log(n))$



Average case

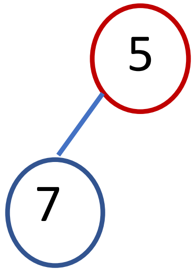
AVL Tree

Time complexity analysis:

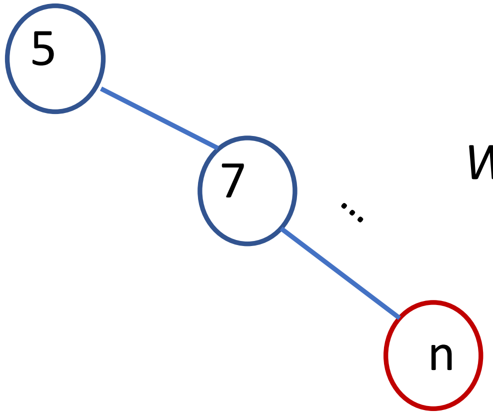
Operation	Best Case	Average Case	Worst Case
Insert	$\Omega(1)$	$\theta(logn)$	$O(logn)$
Search	$\Omega(1)$	$\theta(logn)$	$O(logn)$
Deletion	$\Omega(1)$	$\theta(logn)$	$O(logn)$

Binary Heap Tree

<u>Time complexity analysis:</u>



Best case

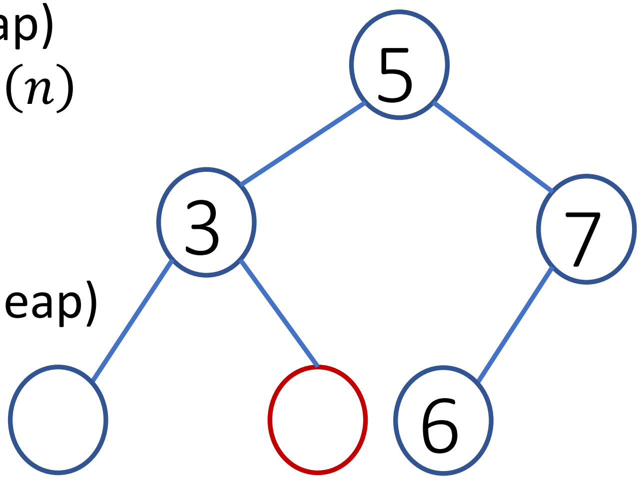


Worst case

- Insertion
  - ✓ Best case : *<u>Insertion nearer to the root node</u>* → $\Omega(1)$
  - ✓ Worst case : *<u>Insertion heap tree</u>* → $O(log(n))$
  - ✓ Average case : *<u>Insertion heap tree</u>* → $\theta(log(n))$

- Search
  - ✓ Best case : *<u>Found at the root node</u>* → $\Omega(1)$ (Max-heap or Min-heap)
  - ✓ Worst case : *<u>Searching the last leaf node in the heap tree takes</u>* → $O(n)$
  - ✓ Average case : *<u>Searching for n^{th} node in the heap tree</u>* → $\theta(n)$

- Deletion
  - ✓ Best case : *<u>Deletion at the root node</u>* → $\Omega(1)$ (Max-heap or Min-heap)
  - ✓ Worst case : *<u>Leaf node of any heap tree</u>* → $O(log(n))$
  - ✓ Average case : *<u>Deletion of a node in a heap tree</u>* → $\theta(log(n))$



Average case

Binary Heap Tree

Time complexity analysis:

Operation	Best Case	Average Case	Worst Case
Insert	$\Omega(1)$	$\theta(logn)$	$O(logn)$
Search	$\Omega(1)$	$\theta(n)$	$O(n)$
Deletion	$\Omega(1)$	$\theta(logn)$	$O(logn)$

thank you!

email:
k.kondepu@iitdh.ac.in