



Sunbeam Institute of Information Technology
Pune and Karad

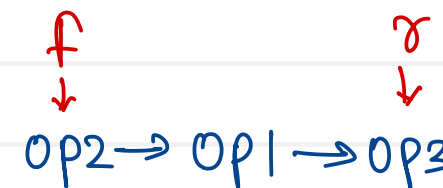
Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

prefix = + 20 40

```
parseInt();
```



Valid Parantheses

Given a string *s* containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: *s* = "()"

Output: true

Example 2:

Input: *s* = "()[]{}"

Output: true

Example 3:

Input: *s* = "([]"

Output: false

Example 4:

Input: *s* = "([)]"

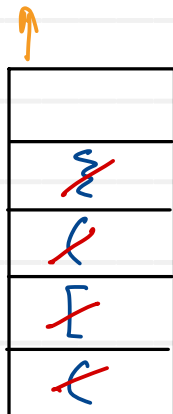
Output: true

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

Parenthesis balancing using stack

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

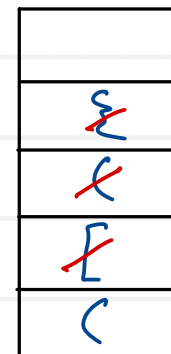
$] == [$
 $\} == \{$
 $) == ($
 $) \neq ($



stack

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

$] == [$
 $\} == \{$
 $] \neq ($



stack

opening

([{
0	1	2

closing

)]	}
0	1	2

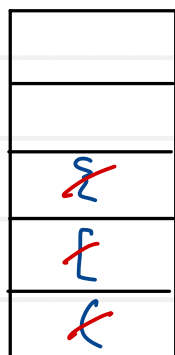
String

↓
indexOf()

↓
returns index of char
returns -1 if char
not found

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

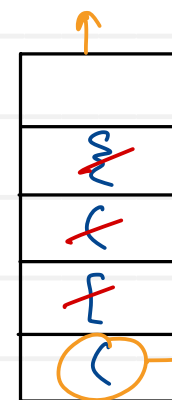
$] == [$
 $\} == \{$
 $) == ($
 $) == ?$



stack

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

$] == [$
 $\} == \{$
 $) == ($



stack

Remove all adjacent duplicates in string

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

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

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

Example 1:

Input: $s = \text{"abbaca"}$

Output: "ca"

a
c
b
a

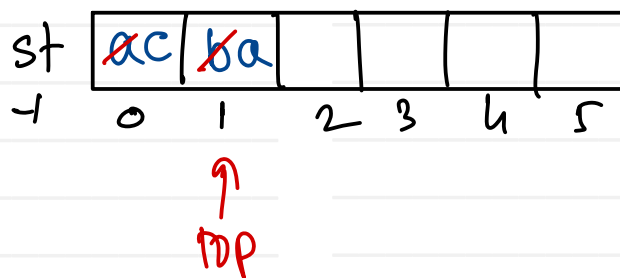
Example 2:

Input: $s = \text{"azxxzy"}$

Output: "ay"

y
x
z
a

abbaca



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

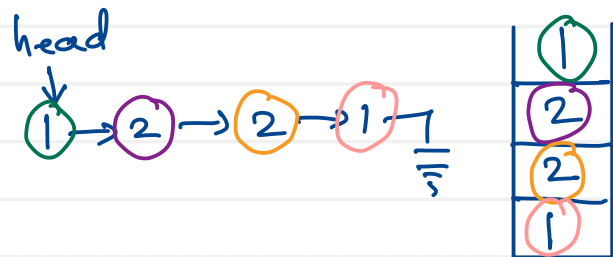
Palindrome Linked List

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

Example 1:

Input: head = [1,2,2,1]

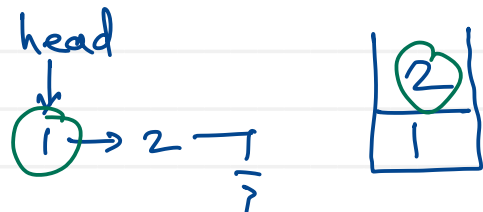
Output: true



Example 2:

Input: head = [1,2]

Output: false



```
boolean isPalindrome(Node head) {
    Stack<Integer> st = new Stack<>();
    Node trav = head;
    while (trav != null) {
        st.push(trav.data);
        trav = trav.next;
    }
    Node trav = head;
    while (!st.isEmpty()) {
        if (trav.data != st.pop())
            return false;
        trav = trav.next;
    }
    return true;
}
```

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

88	33	66	99	11	77	22	55	14
0	1	2	3	4	5	6	7	8

$key == arr[i]$

77
Key

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

89
Key

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

88	33	66	99	11	77	22	55	14
0	1	2	3	4	5	6	7	8

Key = 88 \rightarrow Best case : $O(1)$

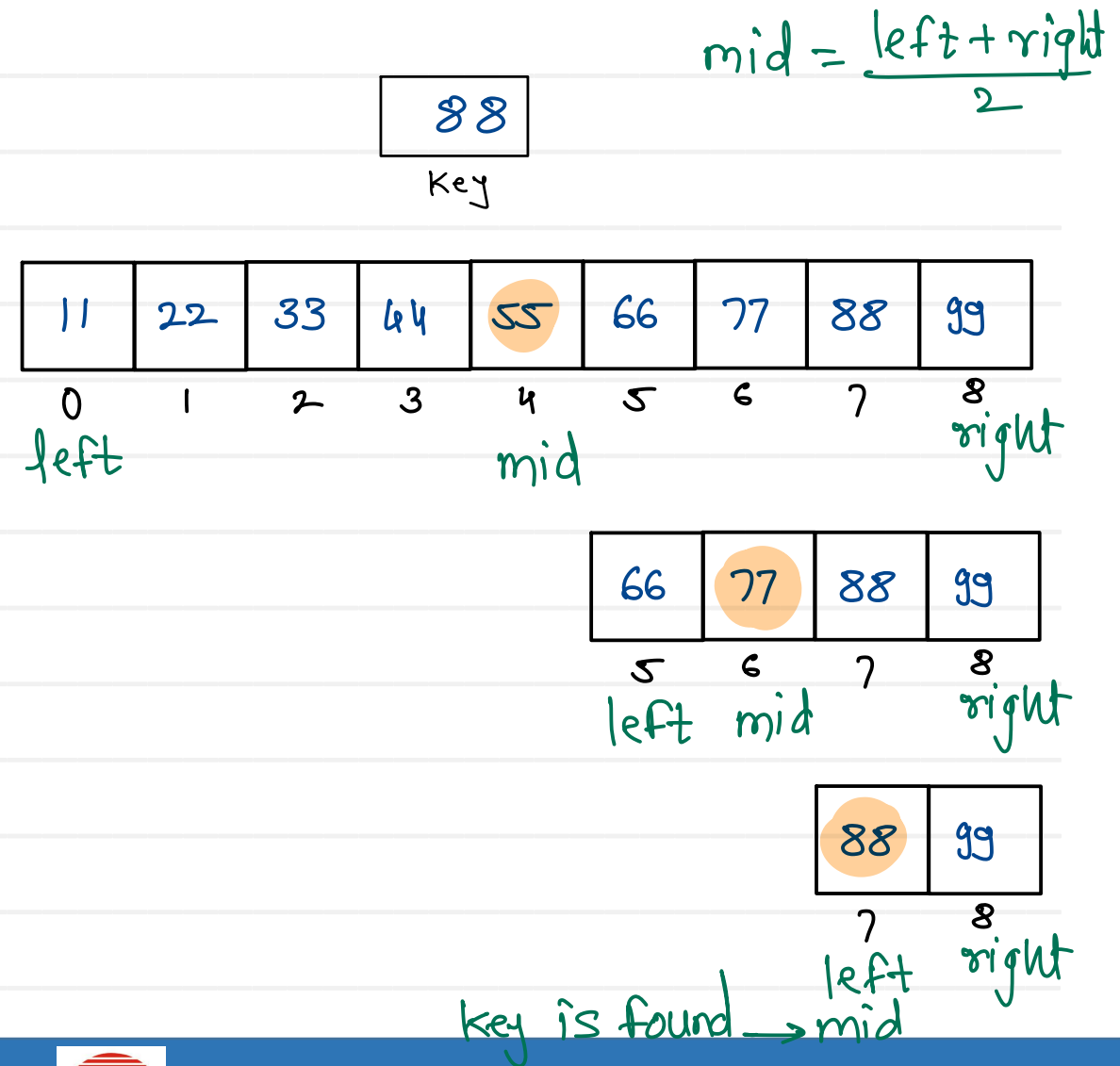
Key = 11 \rightarrow Avg case : $O(n)$

Key = 14/89 \rightarrow Worst case : $O(n)$

$$S(n) = O(1)$$

Binary search

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



Binary search

25
key

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

11	22	33	44
0	1	2	3
left	mid		right

33	44
2	3
left	right
	mid

left partition : $\text{left} \rightarrow \text{mid} - 1$
right partition : $\text{mid} + 1 \rightarrow \text{right}$

valid partition : $\text{left} \leq \text{right}$
invalid partition : $\text{left} > \text{right}$

```

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

```

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

key = 88

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

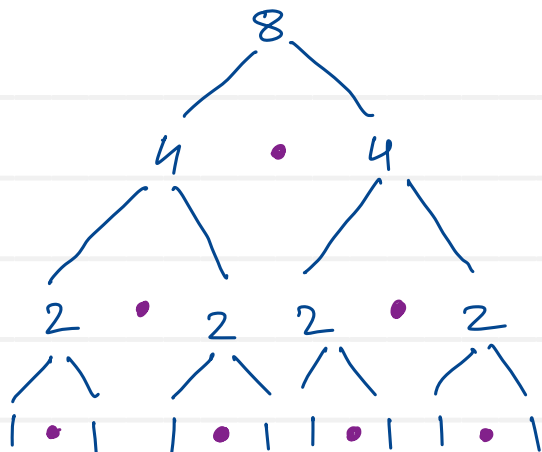
key = 25

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

level 1

level 2

level 3



n - no. of elements

l - no. of level

$$2^l = n$$

$$S(n) = O(1)$$

$$2^l = n$$

$$\log 2^l = \log n$$

$$l \log 2 = \log n$$

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

per level one comparison is done

$$\text{no. of comparisons} = \frac{\log n}{\log 2}$$

$$\text{Time} \propto \frac{\log n}{\log 2}$$

Avg
worst

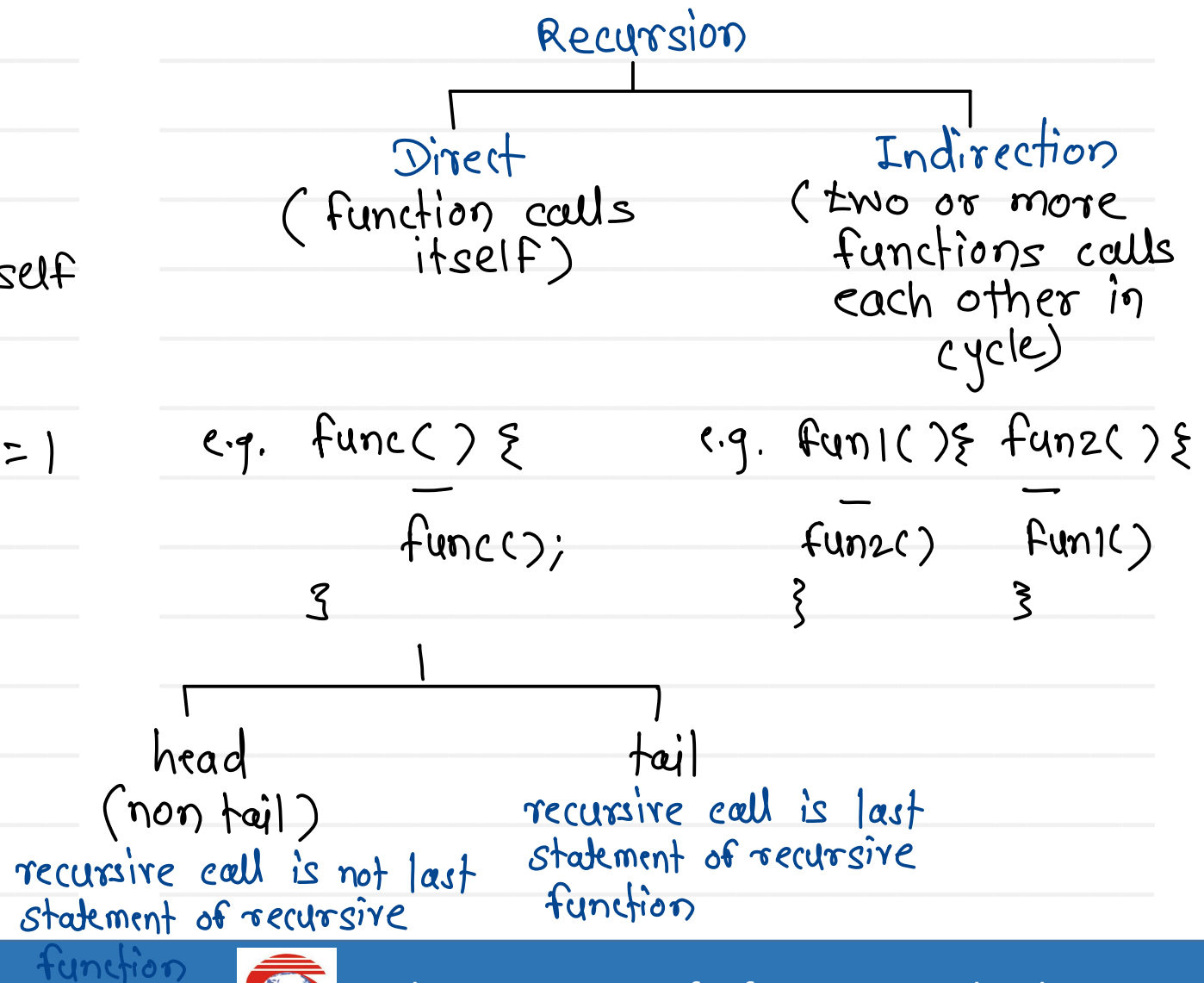
$$T(n) = O(\log n)$$

Best

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

- calling function itself is called as recursion.
- recursion can be used when we know
 - process / formula in terms of itself
 - terminating condition

e.g. $n! = n * (n-1)!$ $0! = 1! = 1$



head recursion

```
void rDisplay (Node trav) {  
    if (trav == null)  
        return;  
    rDisplay (trav.next);  
    sysout (trav.data);  
}
```

tail recursion

```
void fDisplay (Node trav) {  
    if (trav == null)  
        return;  
    sysout (trav.data);  
    fDisplay (trav.next);  
}
```

25
key

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

11	22	33	44
0	1	2	3
l	m		r

2	1	33	44
l	r	2	3
		l	r
		m	

main()
 $\downarrow \uparrow -1$
 binarySearch(arr, 25, 0, 8) $m=4$
 $\swarrow \searrow -1$
 binarySearch(arr, 25, 0, 3) $m=1$
 $\swarrow \searrow -1$
 binarySearch(arr, 25, 2, 3) $m=2$
 $\swarrow \searrow -1$
 binarySearch(arr, 25, 2, 1) \times

Searching algorithms analysis

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

1. Linear search

- Best case - if key is found at few initial locations $\rightarrow O(1)$
- Average case - if key is found at middle locations $\rightarrow O(n)$
- Worst case - if key is found at last few locations/
key is not found $\rightarrow O(n)$

$S(n) = O(1)$

2. Binary search

- Best case - if key is found at first few levels $\rightarrow O(1)$
- Average case - if key is found at middle levels $\rightarrow O(\log n)$
- Worst case - if key is found at last level/
not found $\rightarrow O(\log n)$

$S(n) = O(1)$

Array : linear search - $O(n)$
binary search - $O(\log n)$

Linked List : search - $O(n)$

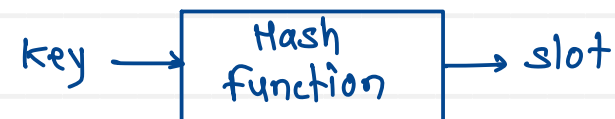
Hashing :
technique which allows to search
data in constant time ($O(1)$)

- implementation of hashing is
called as hash table / hash map

- hashing is a technique in which data can be inserted, deleted and searched in constant average time $O(1)$
- Implementation of hashing is known as hash table
- Hash table is array of fixed size in which elements are stored in key - value pairs

Array - Hash table
Index - Slot
(Associative access)

- In hash table only unique keys are stored
- Every key is mapped with one slot of the table and this is done with the help of mathematical function known as hash function



1. Division method — $k \% m$
2. Multiplication method — $\lfloor m * (k * A) \rfloor$ $0 > A < 1$
3. Mid-square method — k^2 & find middle digit
4. Folding method — divide in two part, sum them, $\% m$
5. Cryptographic hashing — MD5, SHA-1 and SHA-256
6. Universal hashing — $((a * k + b) \% p) \% m$
7. Perfect hashing — collision free hash function for static set of key.
 - guarantees that no two keys will hash to same value

Size = 10

keys values

8-V1

3-V2

10-V3

4-V4

6-V5

13-V6

collision

10, V3	0
	1
	2
3, V2	3
4, V4	4
	5
6, V5	6
	7
8, V1	8
	9

Hash Table

$h(k) = k \% \text{size}$ ← hash function

hash code

$$h(8) = 8 \% 10 = 8$$

$$h(3) = 3 \% 10 = 3$$

$$h(10) = 10 \% 10 = 0$$

$$h(4) = 4 \% 10 = 4$$

$$h(6) = 6 \% 10 = 6$$

$$h(13) = 13 \% 10 = 3$$

insert : $\sim O(1)$

1. slot = h(key)

2. arr[slot] = (key, value)

search : $\sim O(1)$

1. slot = h(key)

2. return arr[slot].value

delete : $\sim O(1)$

1. slot = h(key)

2. arr[slot] = null.

Collision :

situation when two distinct key yield/give same slot.

Collision handling/resolution techniques:

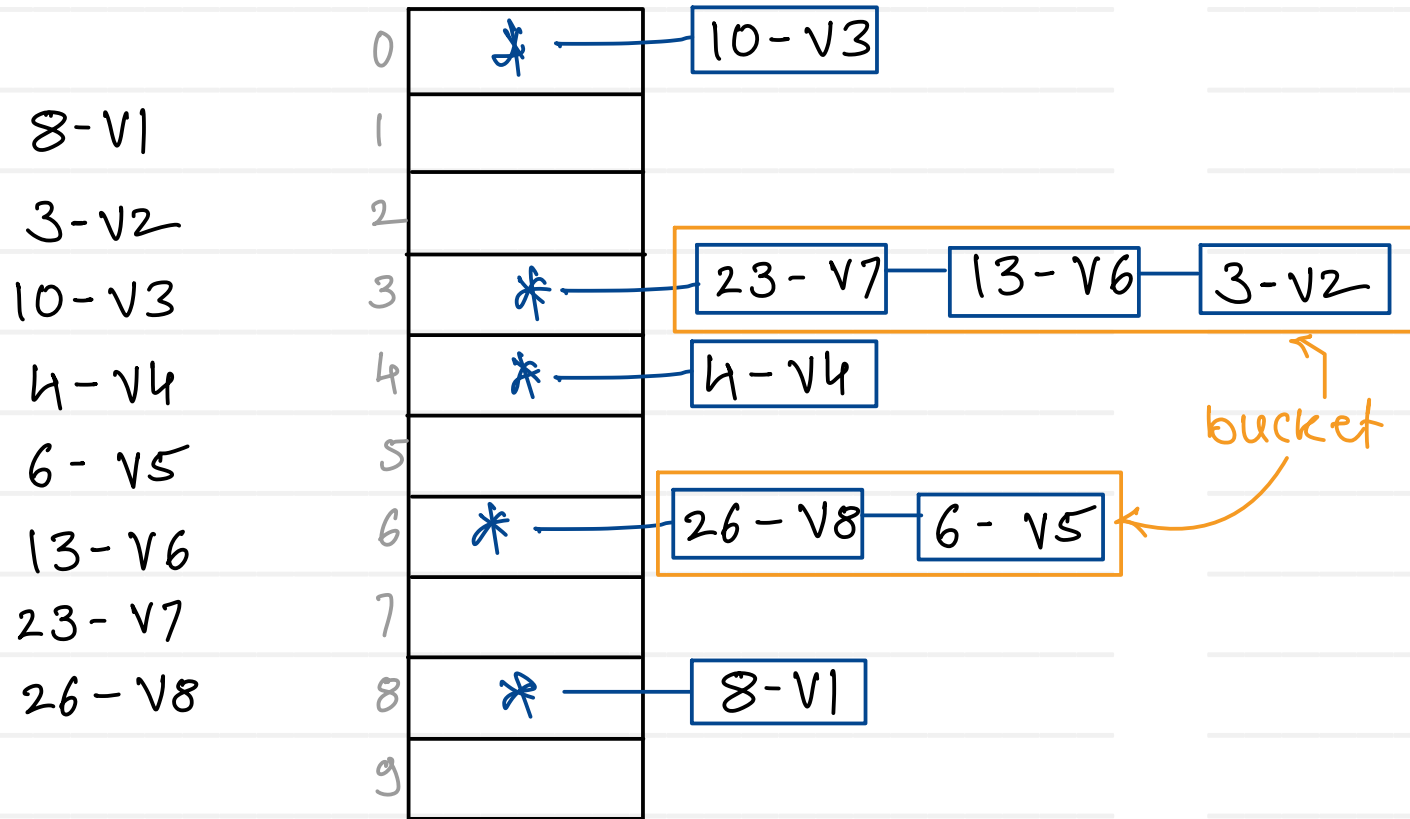
1. Closed addressing
2. open addressing
 - i. linear probing
 - ii. quadratic probing
 - iii. double hashing

Closed Addressing / Chaining / Separate Chaining

per slot one linked list

$$h(k) = k \% \text{ size}$$

Size = 10



Hash Table

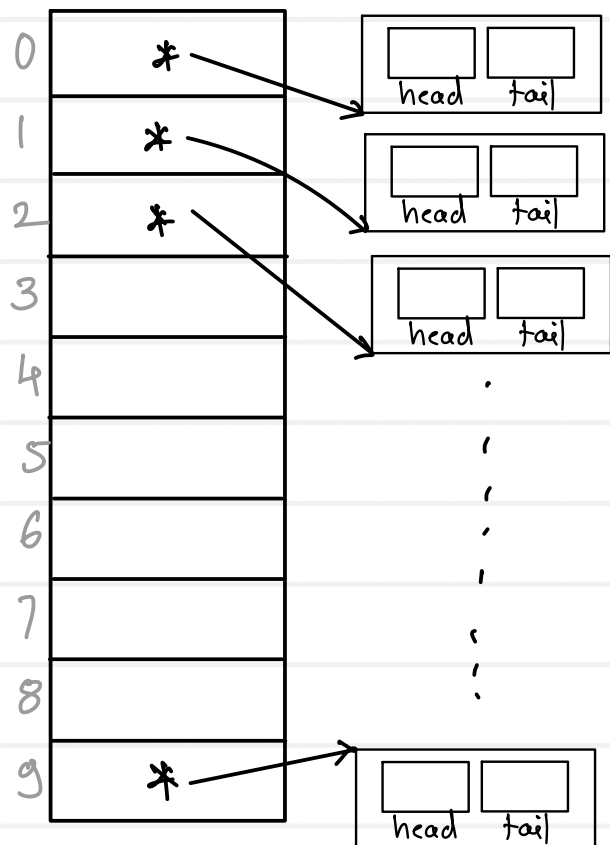
- All buckets other than key position are closed for key-value pair, that why it is called as closed addressing.

Advantage:

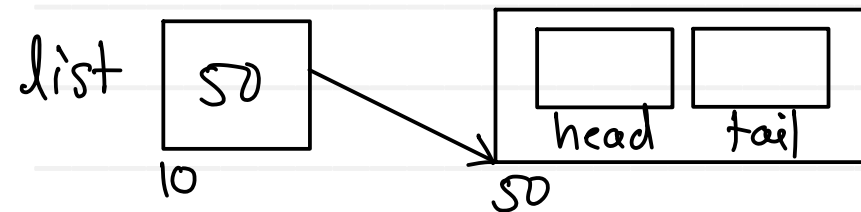
no restriction on number of key value pairs.

Disadvantages:

1. key-value pair are stored outside the table
2. space requirement of this technique is more
3. Worst case time complexity of operations is $O(n)$
↳ if multiple keys yield same slot



```
LinkedList<Integer> list = new LinkedList<>();
```



```
LinkedList<Integer> arr[] = new LinkedList<>[SIZE];
```

or

```
List<Integer> arr[] = new List[SIZE];
```

```
for (i=0; i<SIZE; i++)  
    arr[i] = new LinkedList<>();
```

Open addressing - Linear probing

Size = 10

8-V1

3-V2

10-V3

4-V4

6-V5

13-V6

10, V3	0
	1
	2
3, V2	3
4, V4	4
13, V6	5
6, V5	6
	7
8, V1	8
	9

Hash Table

Probing :

- process of finding next free slot to store key-value pair whenever collision will occur in hash table

$$h(k) = k \% \text{ size}$$

$$h(k, i) = [h(k) + f(i)] \% \text{ size}$$

$$f(i) = i$$

probe numbers

where $i = 1, 2, 3, \dots$

$$h(13) = 13 \% 10 = 3 \text{ (c)}$$

$$1^{\text{st}} \text{ probe : } h(13, 1) = [3 + 1] \% 10 = 4 \text{ (c)}$$

$$2^{\text{nd}} \text{ probe : } h(13, 2) = [3 + 2] \% 10 = 5$$

Rehashing

$$\text{Load factor} = \frac{n}{N}$$

n - number of elements (key-value) present in hash table
N - number of total slots in hash table

e.g. $N = 10, n = 7$

$$\lambda = \frac{7}{10} = 0.7$$



hash table is 70%
filled

- Load factor ranges from 0 to 1.
 - If $n < N$ Load factor < 1 - free slots are available
 - If $n = N$ Load factor $= 1$ - free slots are not available
-
- In rehashing, whenever hash table will be filled more than 60 or 70 % size of hash table is increased by twice
 - Existing key value pairs are remapped according to new size



Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com