Information retrieval (IR)

By: **Dr. LOUNNAS Bilal**

preface

Why study indexation ?

Indexing is an important process in Information Retrieval (IR) systems. It forms the core functionality of the IR process since. Indexing reduces the documents to the informative terms contained in them

Basic

Index ?

In general, index refers to the organization of data according to a specific schema or plan, in order to make information more presentable and accessible.

Example: Database index columns, Term-document.

Basic

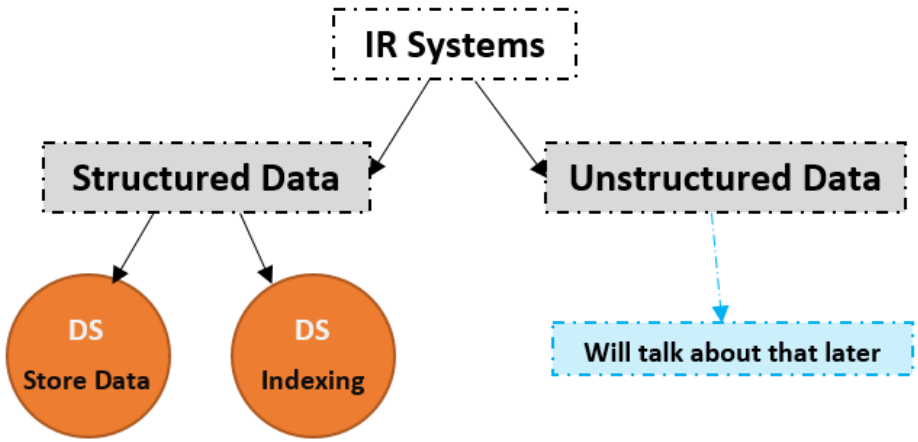
Indexing ?

It's the process of creating indexes, in order to improve the speed of data retrieval operations at the cost of additional writes and storage space to maintain the index data structure.

Example: Microsoft Indexing Service, Web indexing (Search engine).

Introduction

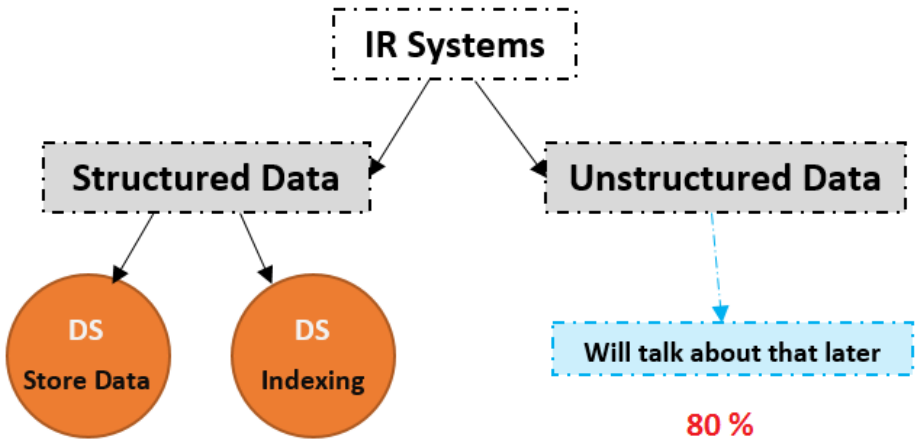
IRs and DS (Data structures)



- **SD** tends to refers to information in "tables" and has a clear, overt semantic structure.
- **UD** does not have a clear, overt semantic structure (eg: free text on a webpage, audio, video).

Introduction

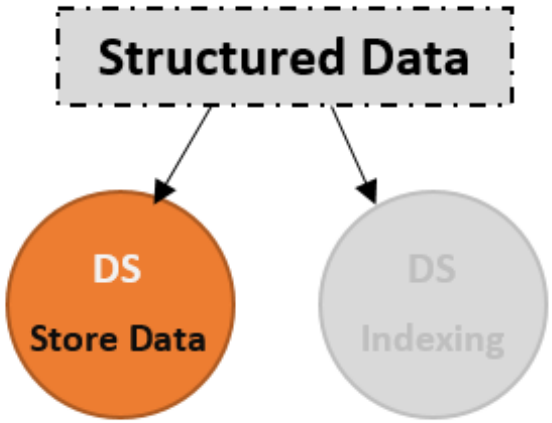
IRs and DS (Data structures)



- **80%** of business-relevant information originates in **unstructured** form, primarily text.
- Emails, Word Processing Files, PDF files, Spreadsheets, Digital Images, Video, Audio, Social Media Posts...etx

Introduction

DS (Store data)

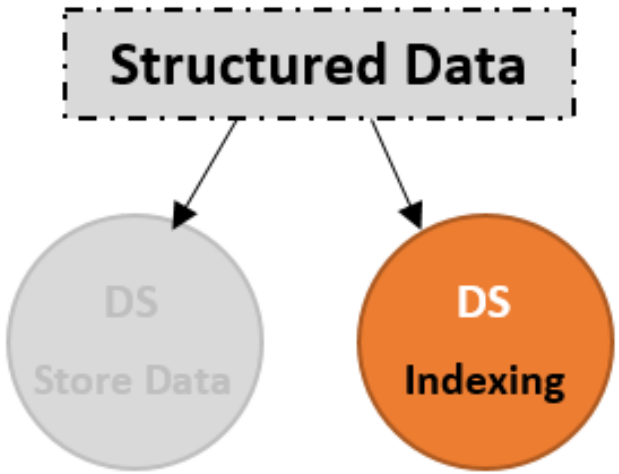


Data structures that store data is an ordinary IRs that have the basics operation (insertion, deletion, updation and most important **searching**).

Example: Array list that hold information of students.

Introduction

DS (Indexing)

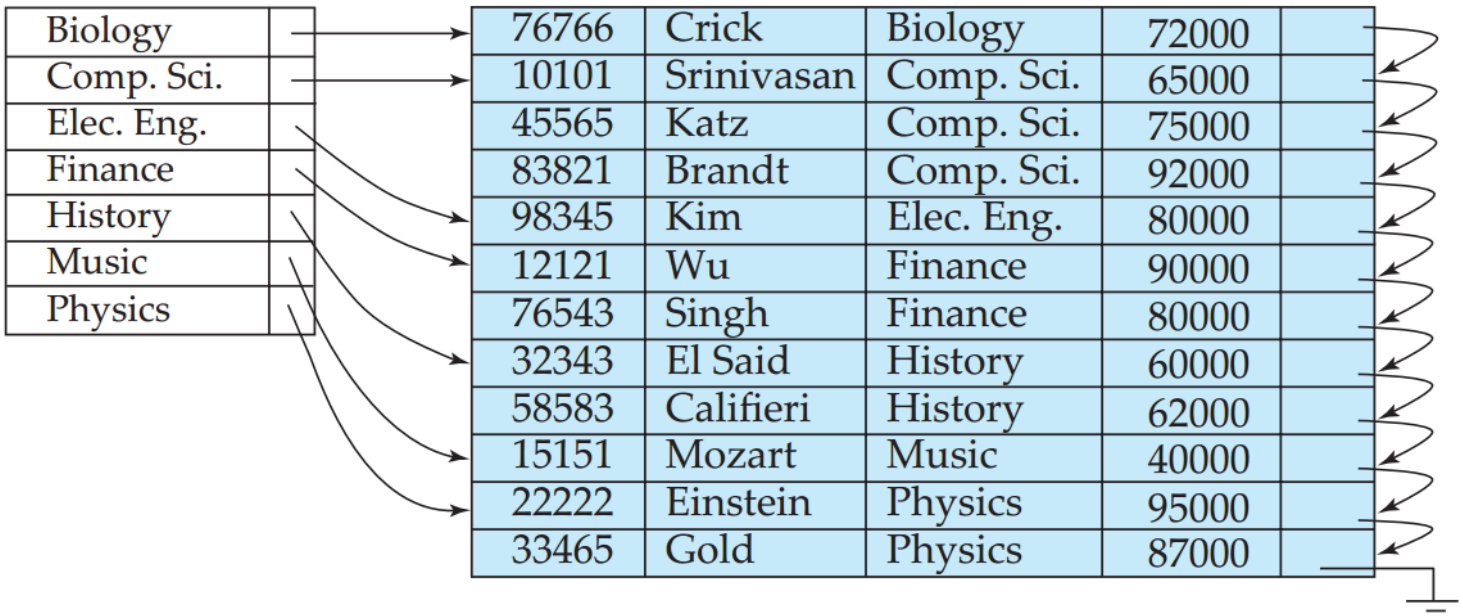


Most of IRs today used data structure to hold an indexer for another data structure that store data.

Example: Student table in SQL server DB, we use a data structure such as Linked List or B Tree to create and store index for student table.

Introduction

DS (Indexing)



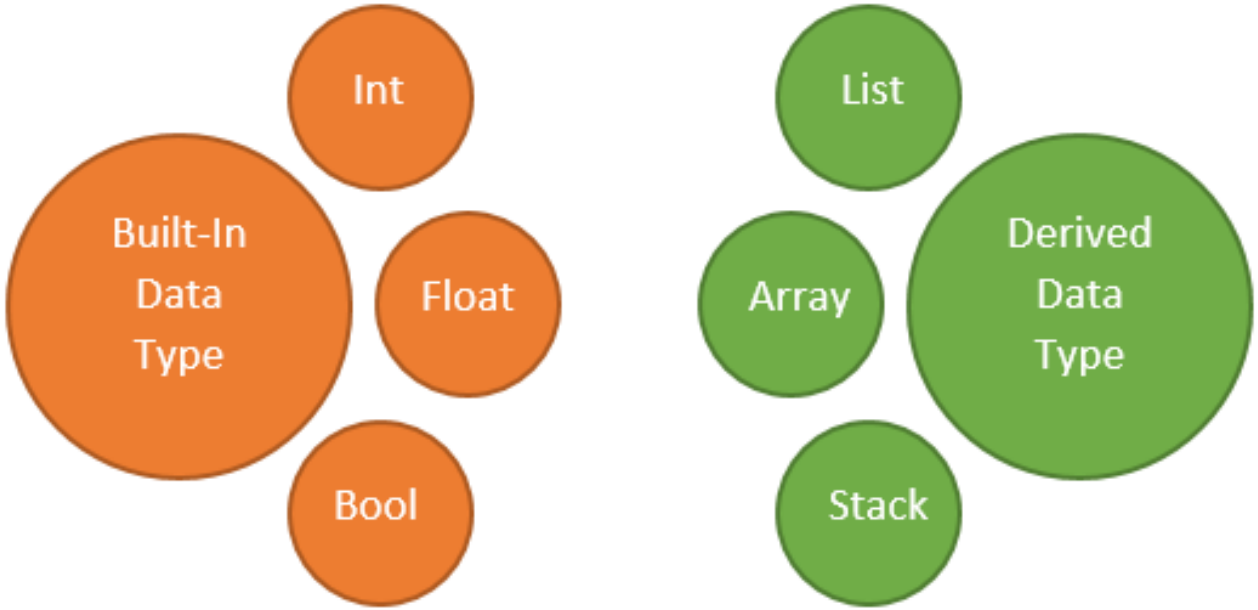
Data definition

Data Definition: defines a particular data with the following characteristics:

- Atomic: Definition should define a single concept.
- Traceable: Definition should be able to be mapped to some data element.
- Accurate: Definition should be unambiguous.
- Clear and Concise: Definition should be understandable.

Data Object represents an object having a data.
Data Type is a way to classify various types of data such as integer, string, etc.

Data types



1. Built-in Data Type
2. Derived Data Type

Basic operations

Operations of data structures

The data in the data structures are processed by certain operations.

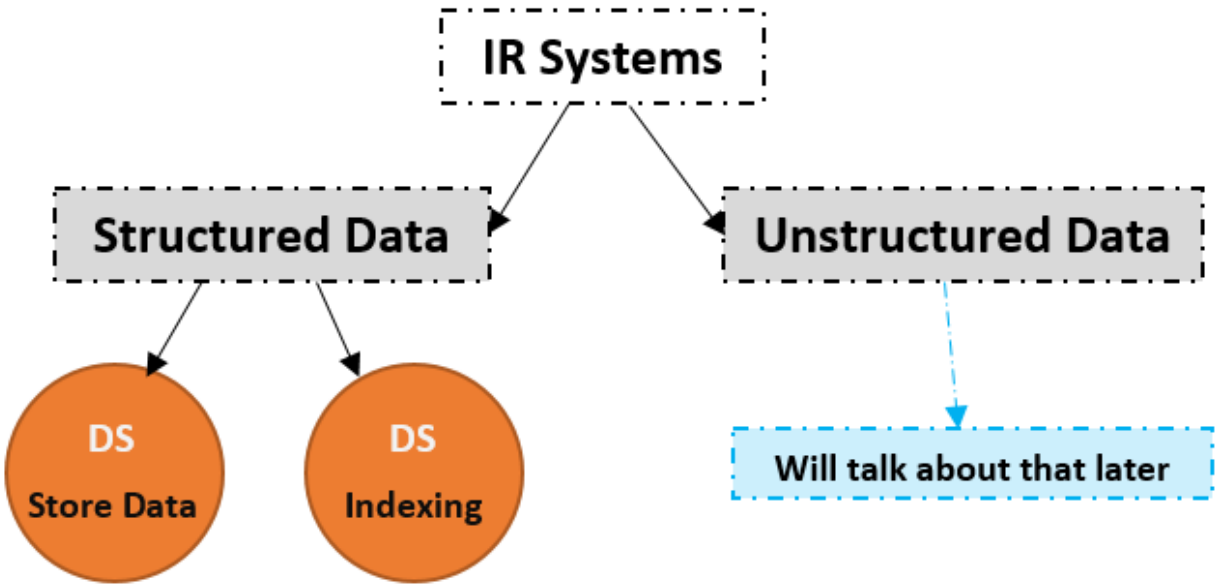
1. Traversing
2. Searching
3. Insertion
4. Deletion
5. Sorting
6. Merging

Classification

The need for algorithms efficiency has led designers to use complex data structures to represent data. Therefor, there are many type of data structure that can be classified as following

...

Classification



Classification

The data structures can also be **classified** on the basis of the following characteristics:

Characterstic
Linear
Non-Linear
Homogeneous
Non-Homogeneous
Static
Dynamic

Table: Characteristics of classification of data structure

IR and Data structures

Data structures are generally based on the effectiveness ability of a computer to fetch and store data at any place in its memory. Therefore, for an IRs to be usable and efficient, it must retrieve data rapidlly and efficiently.

The need for efficiency has led designers to use complex data structures to represent data.

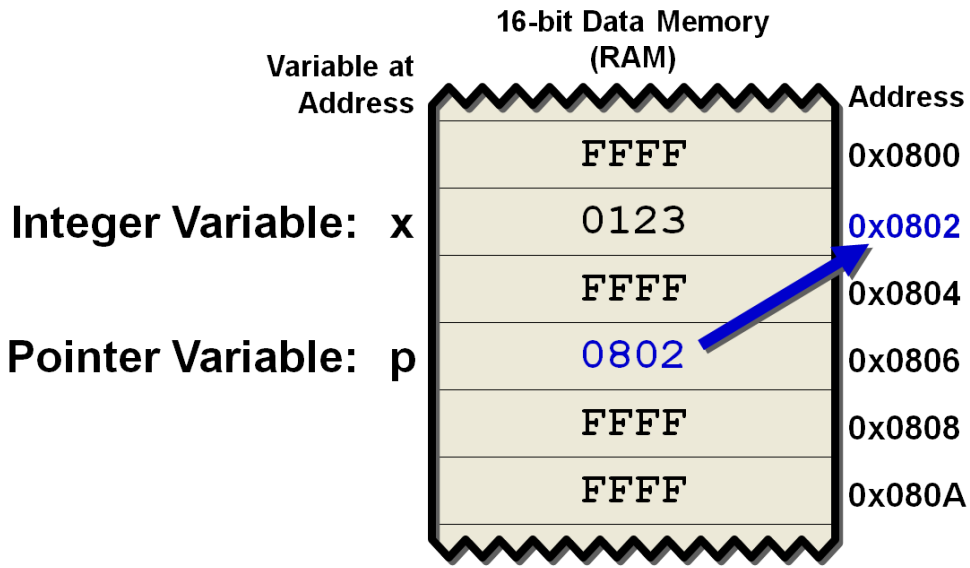
Linked List | Stacks | Queues | Maps | Hash Tables | Binary Tree | Binary Heap | Graph

IR and Data structures

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

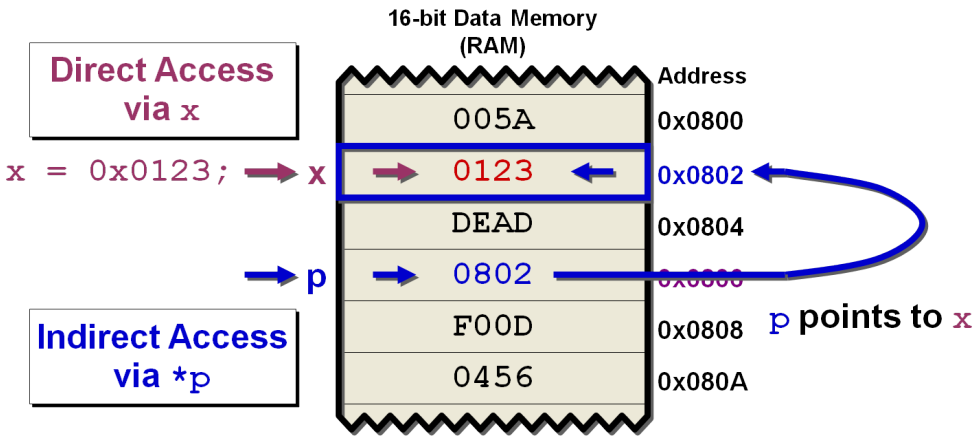
Pointers

In computer science, a pointer is an object, whose value refers to (or "points to") another value stored elsewhere in the computer memory using its memory address. In other means it's a variable which contains the address in memory of another variable. We can have a pointer to any variable type.



Pointers

Example:



Pointers

Advantages of using pointers (the key performance of C)

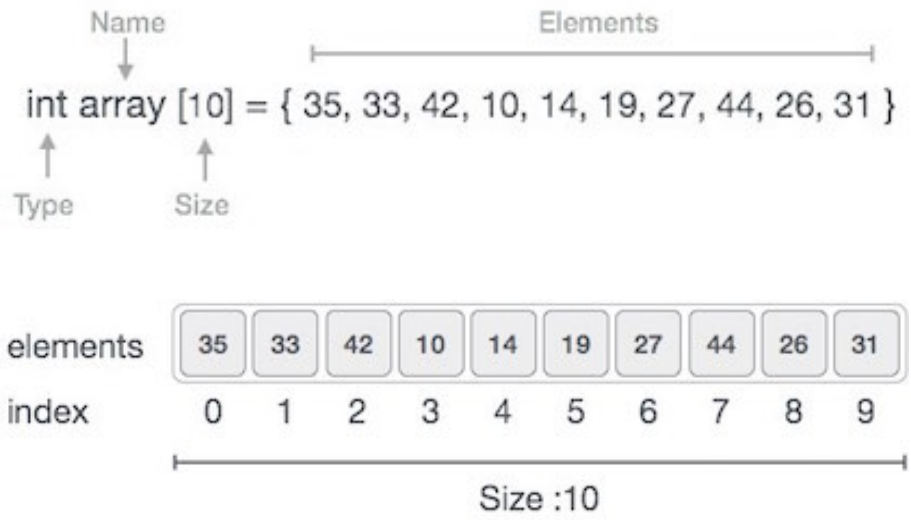
1. Pointers reduce the length and complexity of a program.
2. They increase execution speed.
3. A pointer enables us to access a variable that is defined outside the function.
4. Pointers are more efficient in handling the data tables.
5. Pointers are used to construct different data structures such as linked lists, queues, stacks, etc.

Array

Array is a container which can hold a fix number of items and these items should be of the same type.

1. Element: Each item stored in an array is called an element.
2. Index: Each location of an element in an array has a numerical index, which is used to identify the element.

Array representation



1. Index starts with 0.
2. Array length is 10 which means it can store 10 elements.
3. Each element can be accessed via its index. For example, we can fetch an element at index 6 or 9.

Basic operations

Following are the basic operations supported by an array.

1. Traverse: print all the array elements one by one.
2. Insertion: Adds an element at the given index.
3. Deletion: Deletes an element at the given index.
4. Search: Searches an element using the given index or by the value.
5. Update: Updates an element at the given index.

Insertion operation

01 Insertion operation

Algorithm

Let Array be a linear unordered array of MAX elements.

Let **LA** be a Linear Array (unordered) with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where **ITEM** is inserted into the K^{th} position of **LA**.

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Deletion operation

02 Deletion operation

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the K^{th} position of **LA**.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Search operation

03 Search operation

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of **ITEM** using sequential search.

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

Update operation

04 Update operation

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to update an element available at the K^{th} position of **LA**.

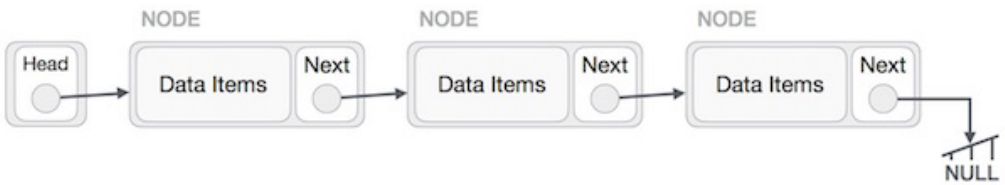
```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

LinkedList

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array.

- Link: Each link of a linked list can store a data called an element.
- Next: Each link of a linked list contains a link to the next link called Next.
- LinkedList: A Linked List contains the connection link to the first link called First.

LinkedList representation



- Linked List contains a link element called Head.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of LinkedList

Types of LinkedList.

Following are the various types of linked list.

1. Simple Linked List: Item navigation is forward only.
2. Doubly Linked List: Items can be navigated forward and backward.
3. Circular Linked List: Last item contains link of the first element as next and the first element has a link to the last element as previous.

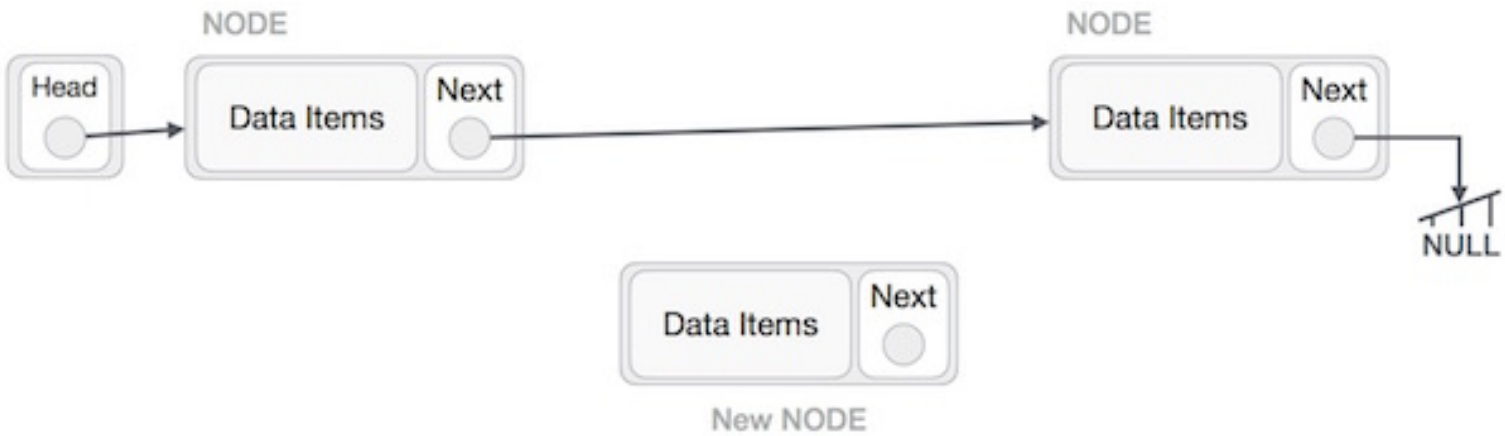
Basic operations

Following are the basic operations supported by a LinkedList.

1. Insertion: Adds an element at the beginning of the list.
2. Deletion: Deletes an element at the beginning of the list.
3. Display: Displays the complete list.
4. Search: Searches an element using the given key.
5. Reverse: Reverse the whole linked list by making the last node as the first node.

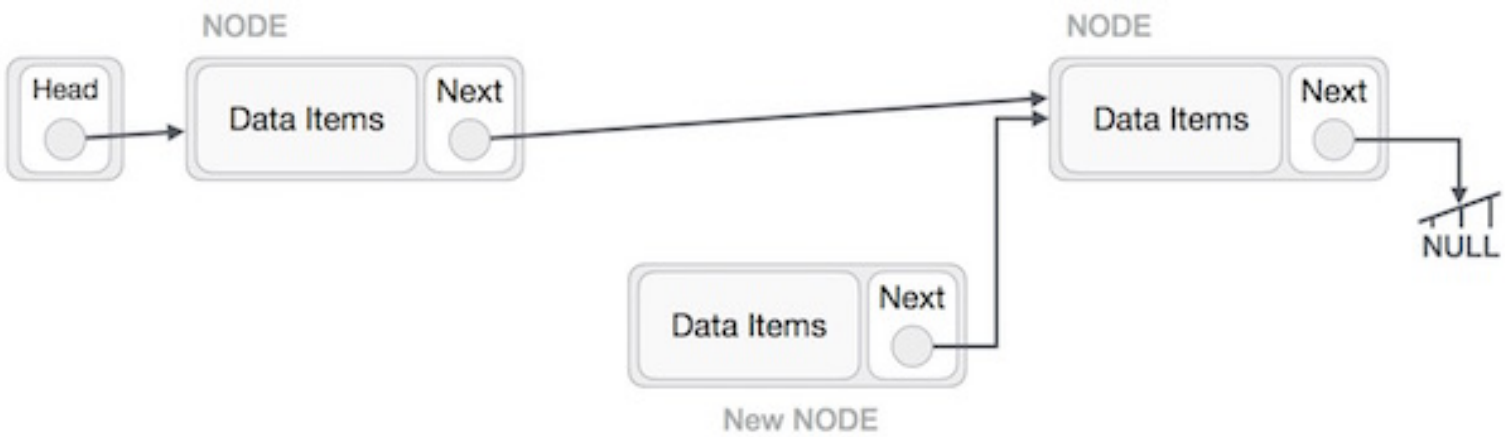
Insertion operation

01 Insertion operation



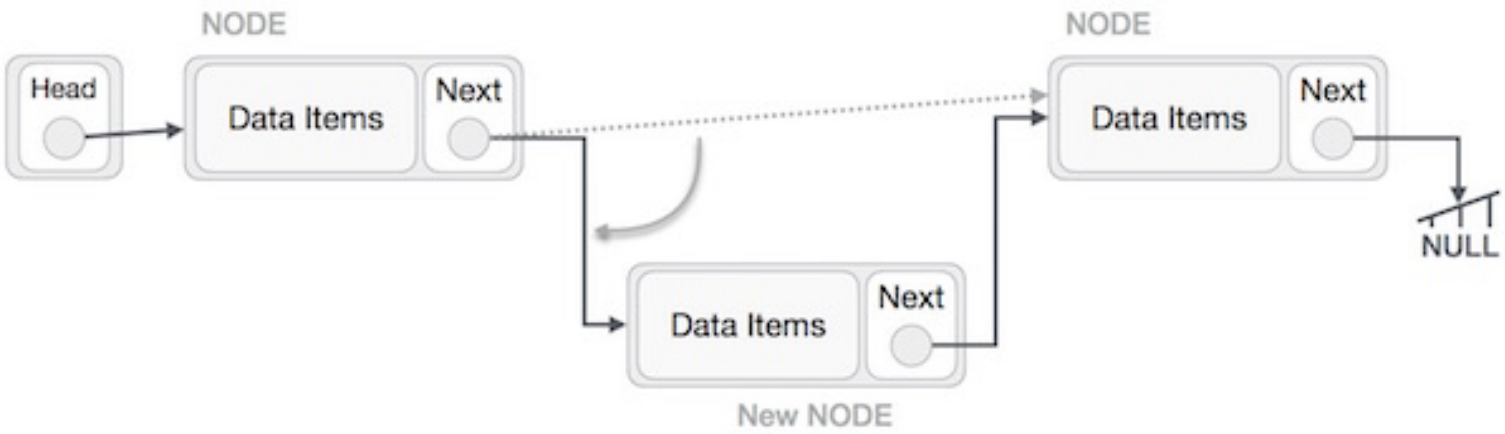
Insertion operation

01 Insertion operation



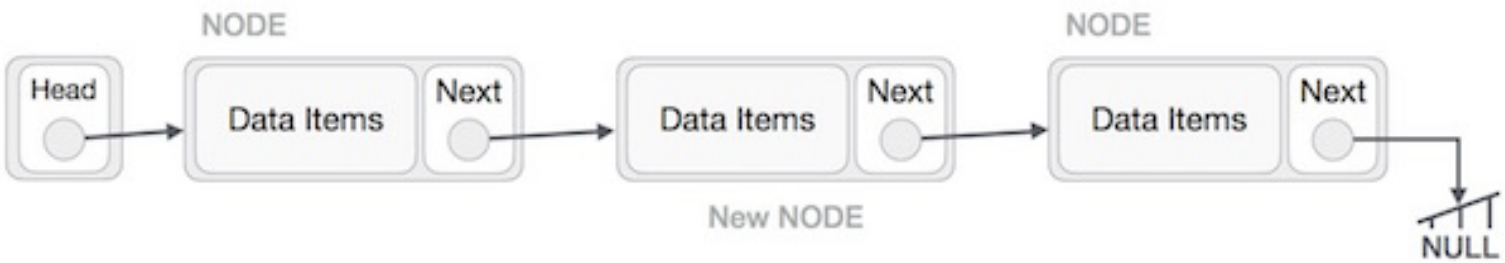
Insertion operation

01 Insertion operation



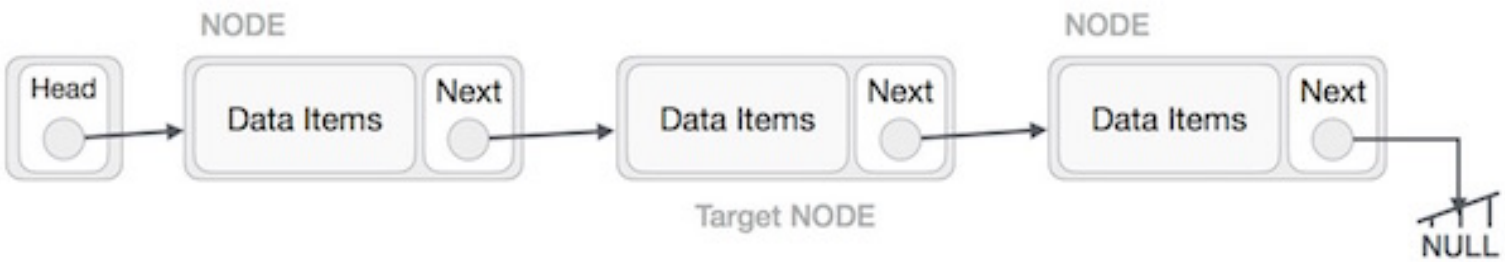
Insertion operation

01 Insertion operation



Deletion operation

02 Deletion operation



Deletion operation

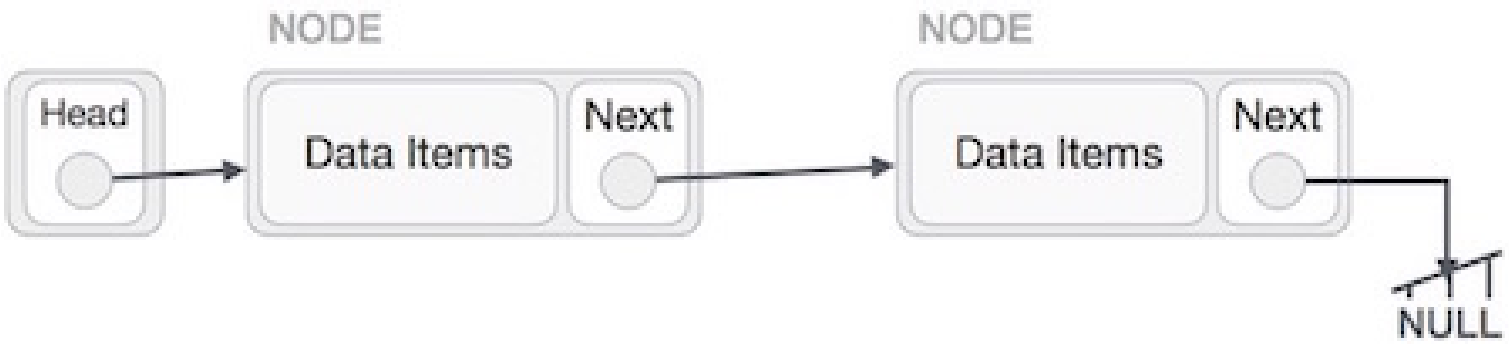
02 Deletion operation

Deletion operation

02 Deletion operation

Deletion operation

02 Deletion operation



LinkedList

Example of **LinkedList**

Seq = AA G CC A C TT C A TT G T G A TT AA C

Seq	A	A	G	C	C	A	C	T	T	C	A	T	T	G	T	G	A	T	T	A	A	C
Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

A_LL = {0,1,5,10,16,19,20};

G_LL = {2, 13, 15};

C_LL = {7, 8, 11 , 12, 14, 17, 18};

T_LL = {3, 4, 6, 9, 21 };

LinkedList

Example of LinkedList

Seq = AA G CC A C TT C A TT G T G A TT AA C

Seq	A	A	G	C	C	A	C	T	T	C	A	T	T	G	T	G	A	T	T	A	A	C
Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

A_LL = {0,1,5,10,16,19,20};

G_LL = {2, 13, 15};

C_LL = {7, 8, 11 , 12, 14, 17, 18};

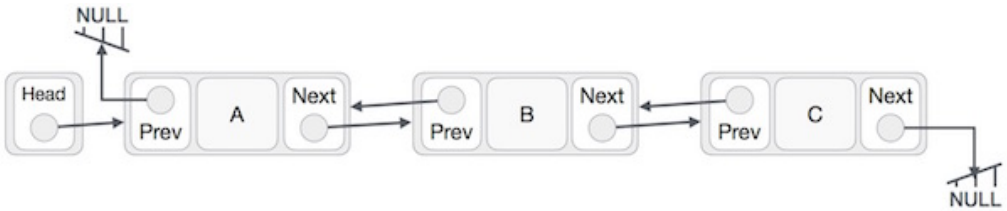
T_LL = {3, 4, 6, 9, 21 };

Doubly LinkedList

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- Link: Each link of a linked list can store a data called an element.
- Next: Each link of a linked list contains a link to the next link called Next.
- Prev: Each link of a linked list contains a link to the previous link called Prev.
- LinkedList: A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly LinkedList representation



- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Basic operations

Following are the basic operations supported by a Doubly LinkedList.

1. Insertion: Adds an element at the beginning of the list.
2. Deletion: Deletes an element at the beginning of the list.
3. Insert Last: Adds an element at the end of the list.
4. Delete Last: Deletes an element from the end of the list.
5. Insert After: Adds an element after an item of the list.
6. Delete: Deletes an element from the list using the key.
7. Display forward: Displays the complete list in a forward manner.
8. Display backward: Displays the complete list in a backward manner.

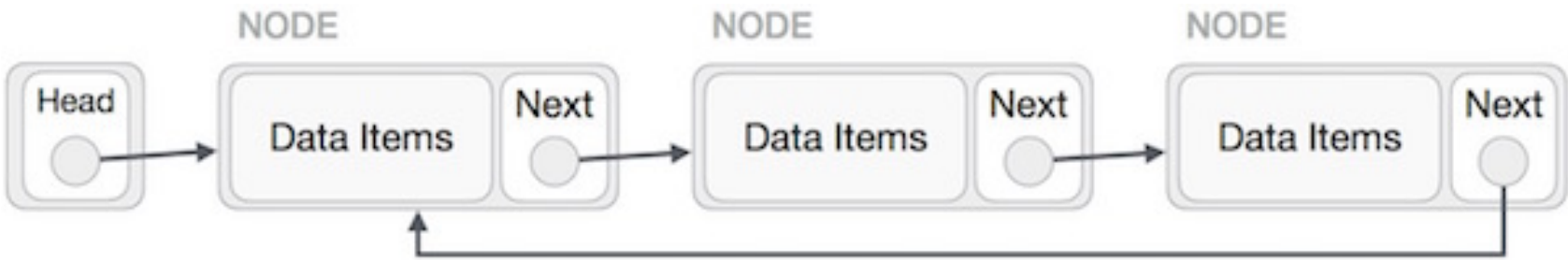
Doubly LinkedList

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

- Link: Each link of a linked list can store a data called an element.
- Next: Each link of a linked list contains a link to the next link called Next.
- Prev: Each link of a linked list contains a link to the previous link called Prev.
- LinkedList: A Linked List contains the connection link to the first link called First and to the last link called Last.

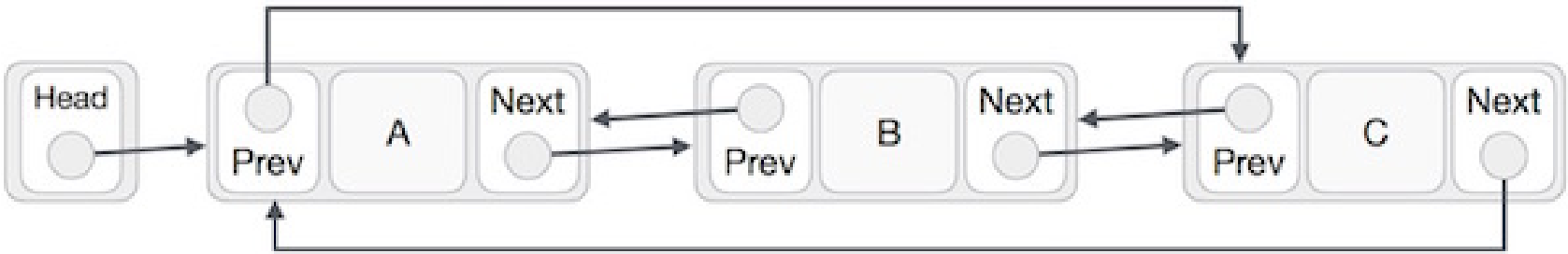
Basic operations

Singly LinkedList as Circular.



Basic operations

Doubly LinkedList as Circular.

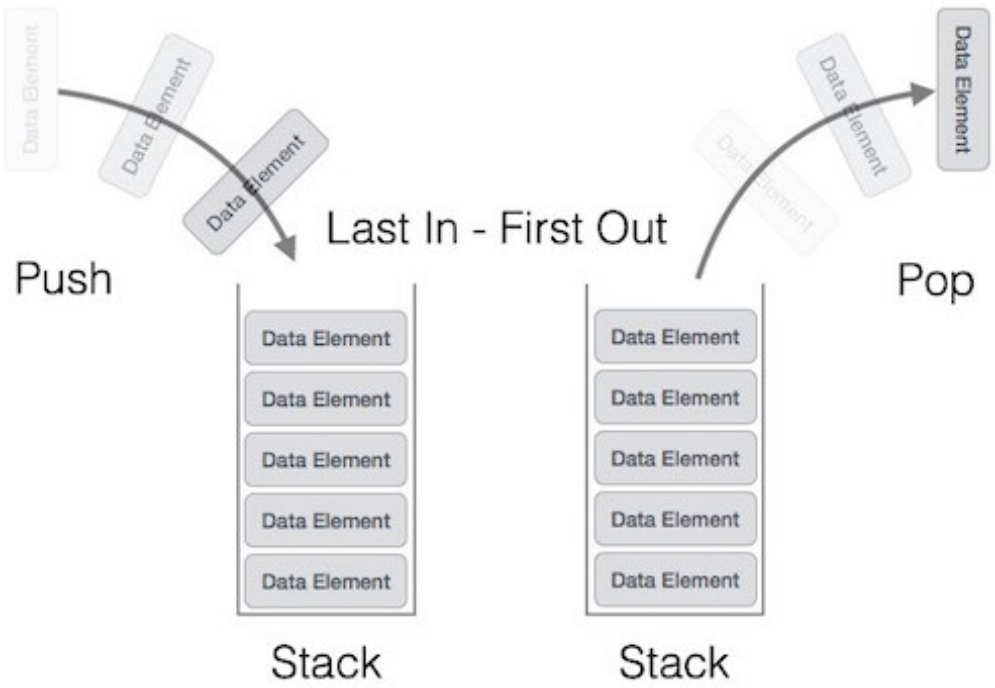


Stack

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example: a deck of cards or a pile of plates, etc.

Stack strategy

LIFO - Last-In-First-Out



- Stack strategy is LIFO.
- Two operation PUSH, POP.
- Fixed size (Array)
- Dynamic size (LinkedList)

Basic operations

Following are the basic operations supported by a Stack.

1. **push()**: pushing (storing) an element on the stack.
2. **pop()**: removing (accessing) an element from the stack.
3. peek(): get the top data element of the stack, without removing it.
4. isFull(): check if stack is full.
5. isEmpty(): check if stack is empty.

Peek operation

01 Peek operation

```
begin procedure peek

   return stack[top]

end procedure
```

isFull operation

02 isFull operation

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

isEmpty operation

03 isEmpty operation

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

Push operation

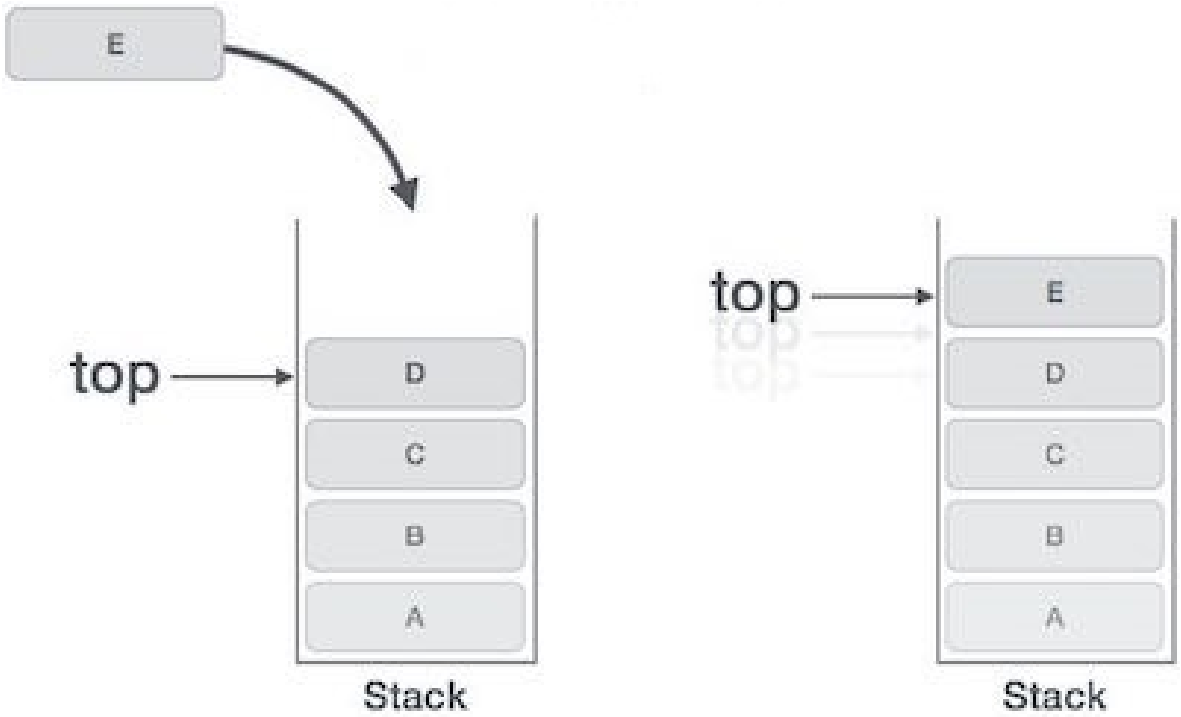
Push operation involves a series of steps:

1. Step 1: Checks if the stack is full.
2. Step 2: If the stack is full, produces an error and exit.
3. Step 3: If the stack is not full, increments top to point next empty space.
4. Step 4: Adds data element to the stack location, where top is pointing.
5. Step 5: Returns success.

If stack is dynamic size implmentation: Step 2 and 3 will be changed

Push operation presentation

Push operation



Push operation

04 Push operation

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1

   stack[top] ← data

end procedure
```

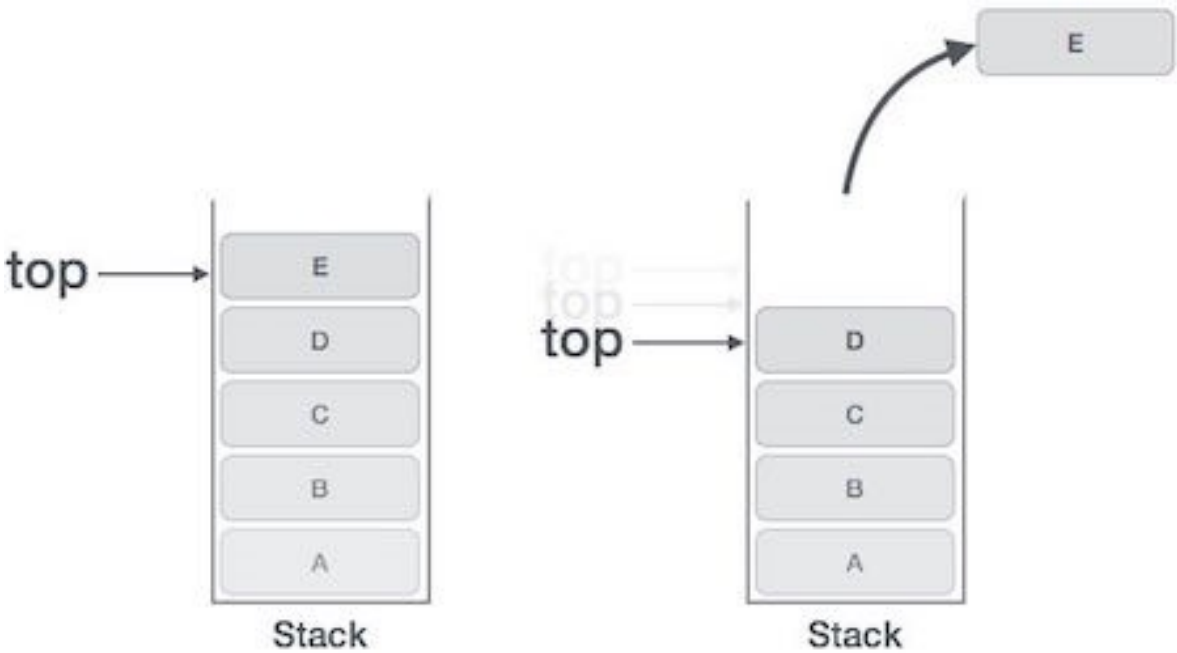
Pop operation

Pop operation involves a series of steps:

1. Step 1: Checks if the stack is empty.
2. Step 2: If the stack is empty, produces an error and exit.
3. Step 3: If the stack is not empty, accesses the data element at which top is pointing.
4. Step 4: Decreases the value of top by 1.
5. Step 5: Returns success.

Pop operation presentation

Pop operation



Pop operation

05 Pop operation

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]

   top ← top - 1

   return data

end procedure
```

Queue

Queue similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

Queue strategy

FIFO - First-In-First-Out

- Queue strategy is LIFO.
- Two operation Enqueue, Dequeue.
- Fixed size (Array)
- Dynamic size (LinkedList)

Basic operations

Following are the basic operations supported by a Queue.

1. **enqueue()**: add (store) an item to the queue.
2. **dequeue()**: remove (access) an item from the queue.
3. peek(): Gets the element at the front of the queue without removing it.
4. isfull(): Checks if the queue is full.
5. isempty(): Checks if the queue is empty.

Peek operation

01 Peek operation

```
begin procedure peek

   return queue[front]

end procedure
```

isFull operation

02 isFull operation

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

isEmpty operation

03 isEmpty operation

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

Enqueue operation

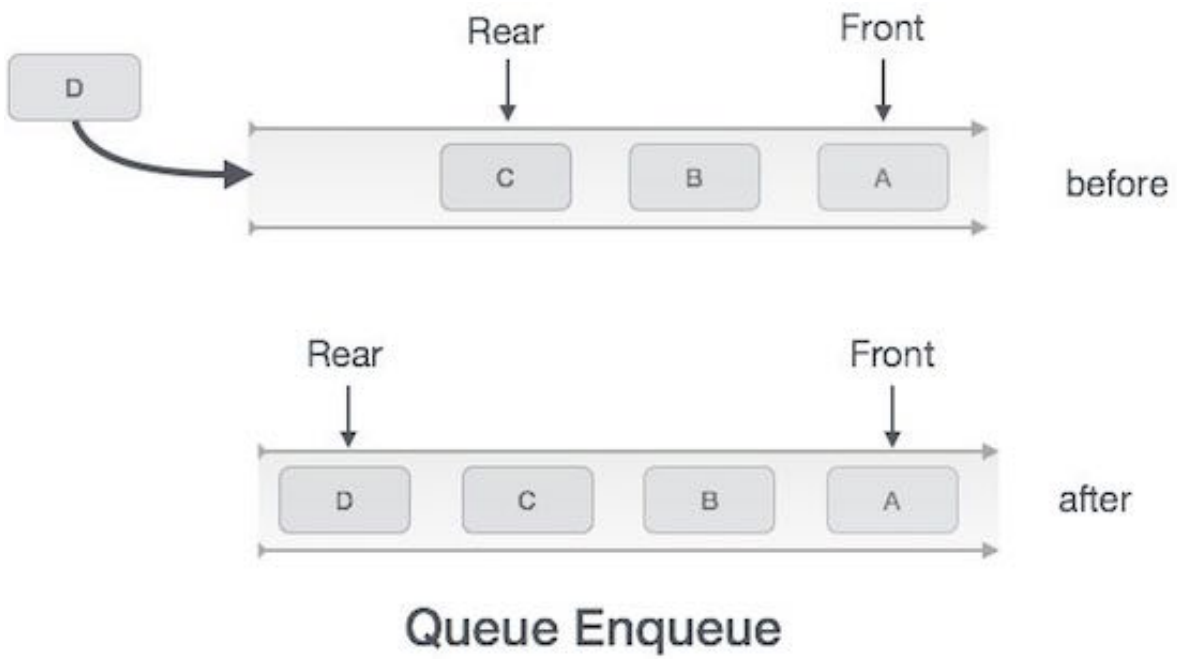
Enqueue operation involves a series of steps:

1. Step 1: Check if the queue is full.
2. Step 2: If the queue is full, produce overflow error and exit.
3. Step 3: If the queue is not full, increment rear pointer to point the next empty space.
4. Step 4: Add data element to the queue location, where the rear is pointing.
5. Step 5: return success.

If queue is dynamic size implmentation: Step 2 and 3 will be changed

Enqueue operation presentation

Enqueue operation



Queue Enqueue

Enqueue operation

04 Enqueue operation

```
procedure enqueue(data)
   if queue is full
      return overflow
   endif

   rear ← rear + 1

   queue[rear] ← data

   return true

end procedure
```

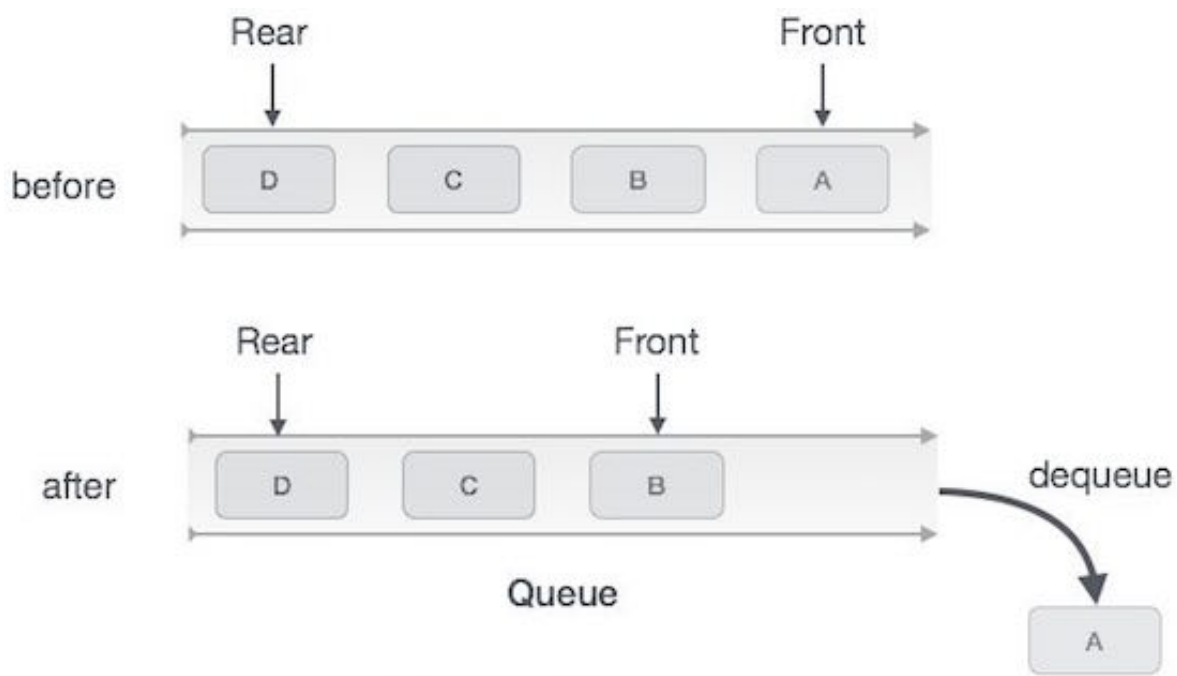
Dequeue operation

Dequeue operation involves a series of steps:

1. Step 1: Check if the queue is empty.
2. Step 2: If the queue is empty, produce underflow error and exit.
3. Step 3: If the queue is not empty, access the data where front is pointing.
4. Step 4: Increment front pointer to point to the next available data element.
5. Step 5: Return success.

Dequeue operation presentation

Dequeue operation



Queue Dequeue

Dequeue operation

05 Dequeue operation

```
procedure dequeue
   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1

   return true
end procedure
```

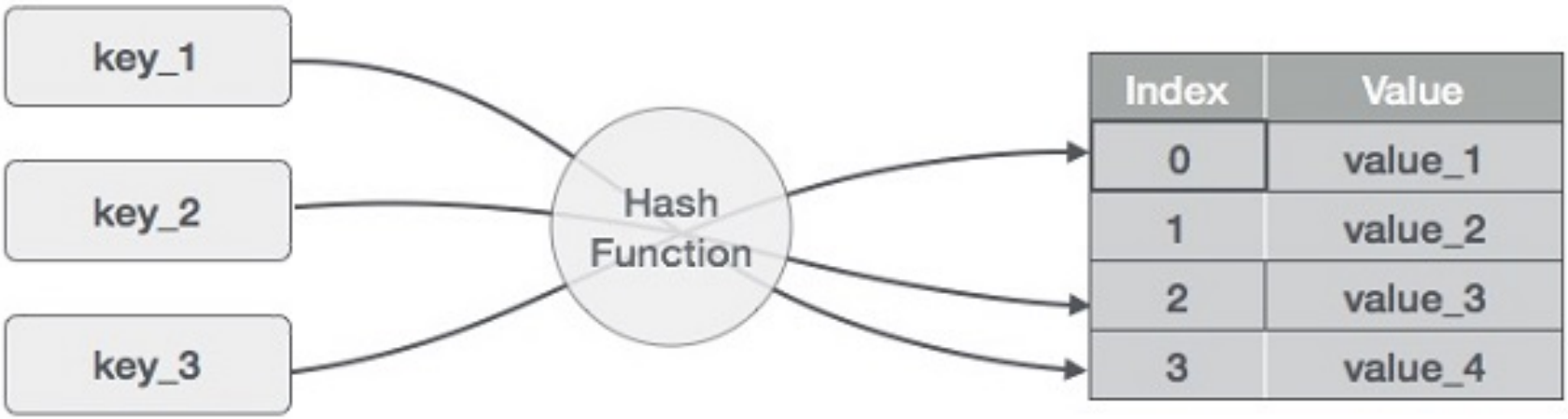
Hash table

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

1. Hash table becomes a DS in which insertion and search operations are very fast irrespective of the size of the data.
2. Hash table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

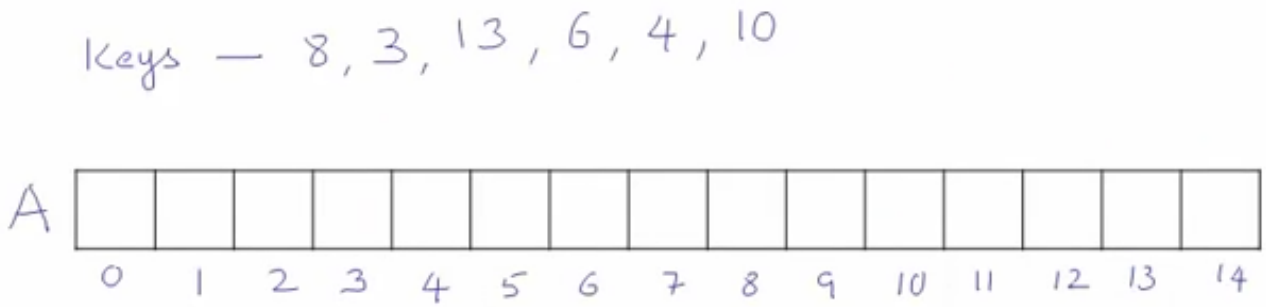
Hashing

Hashing function

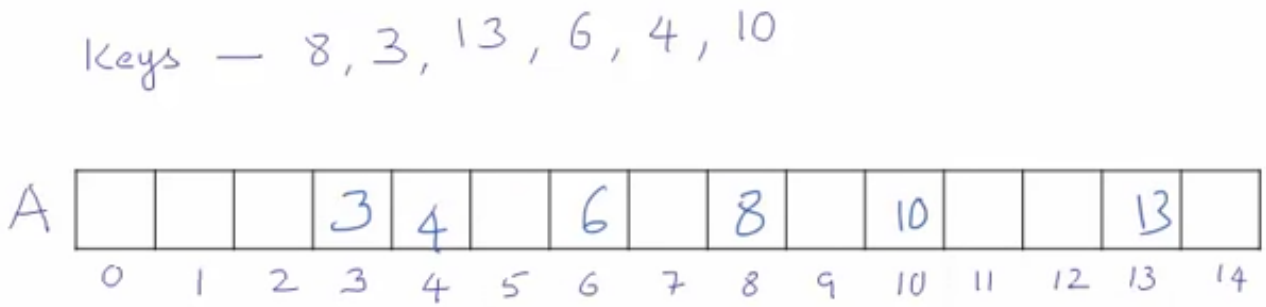


Hashing is a technique to convert a range of key values into a range of indexes of an array

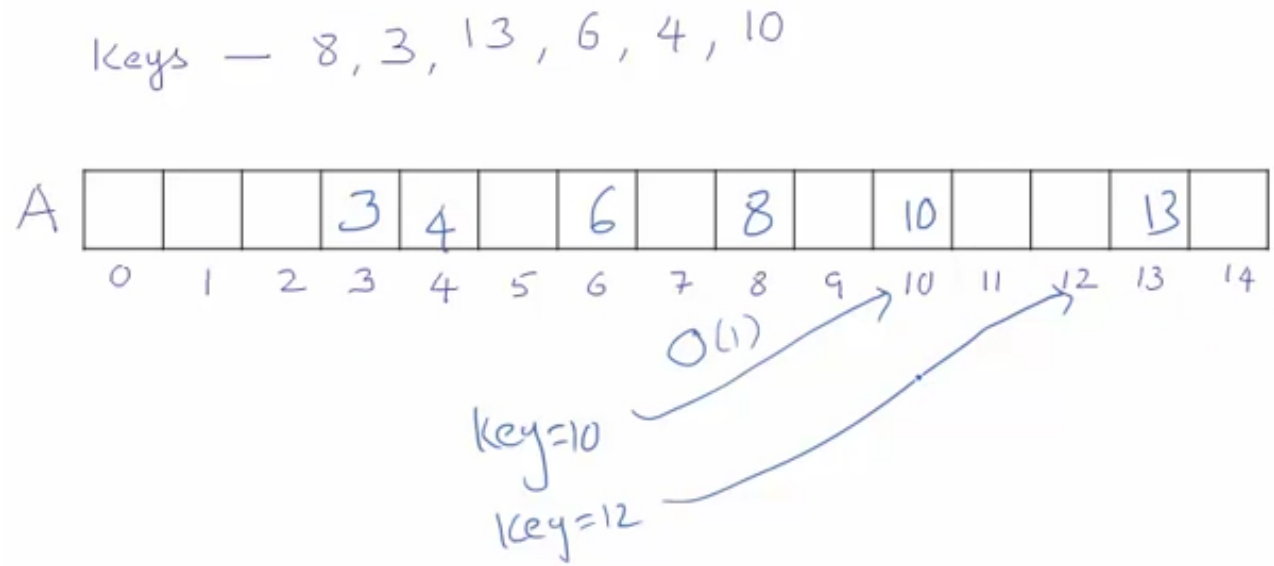
Hashing example



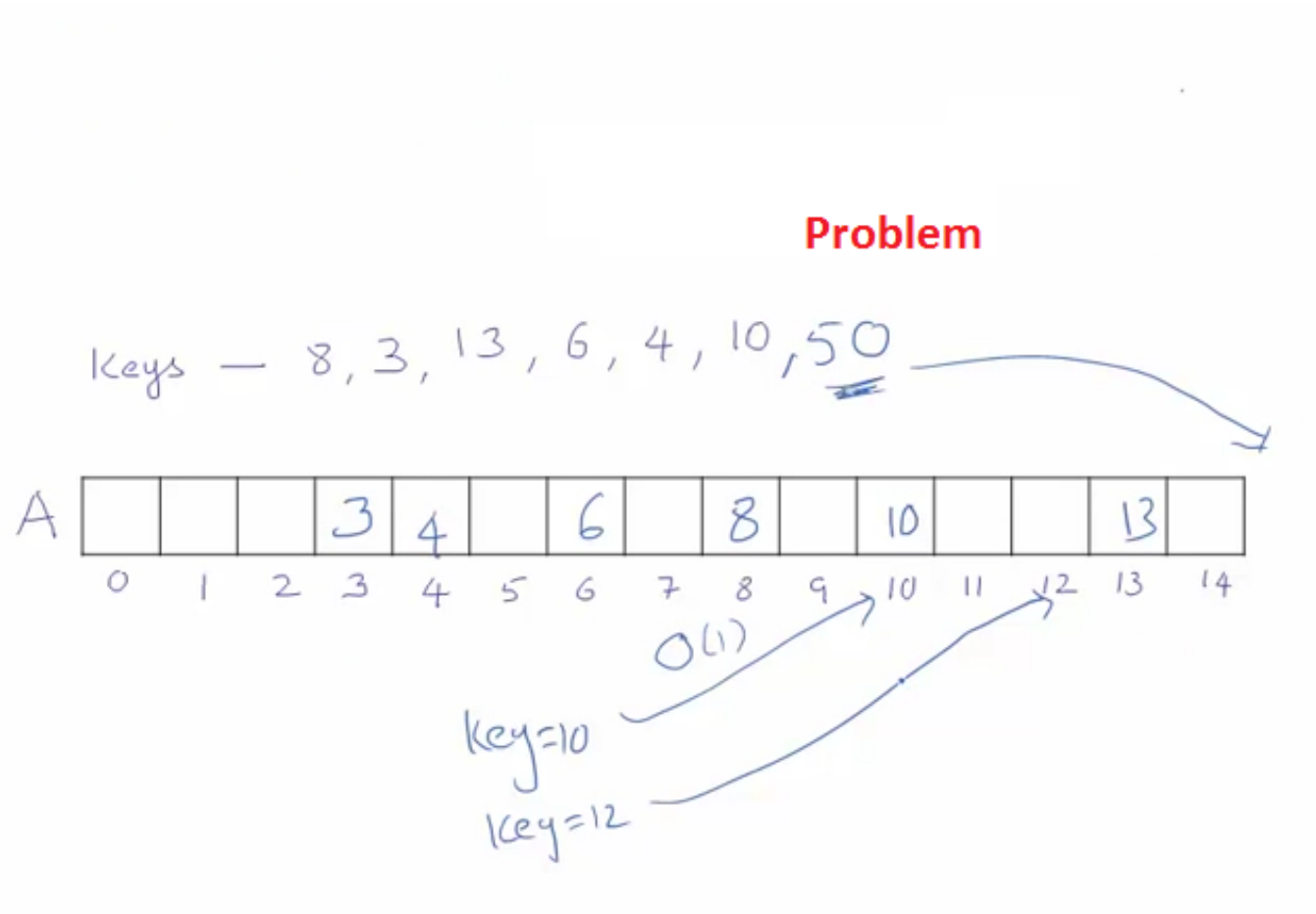
Hashing example



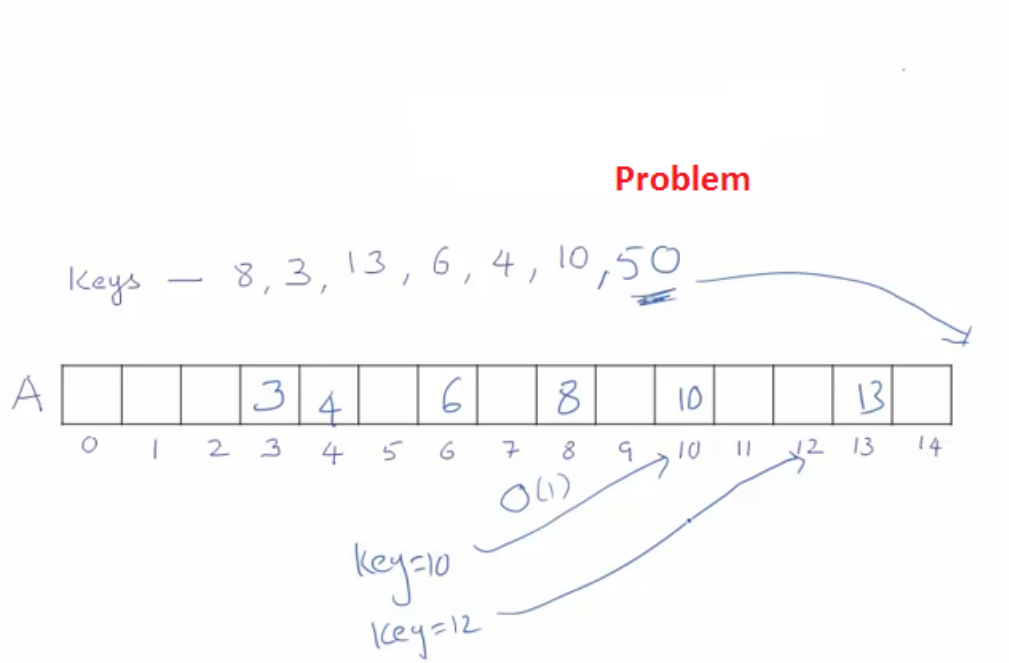
Hashing example



Hashing example

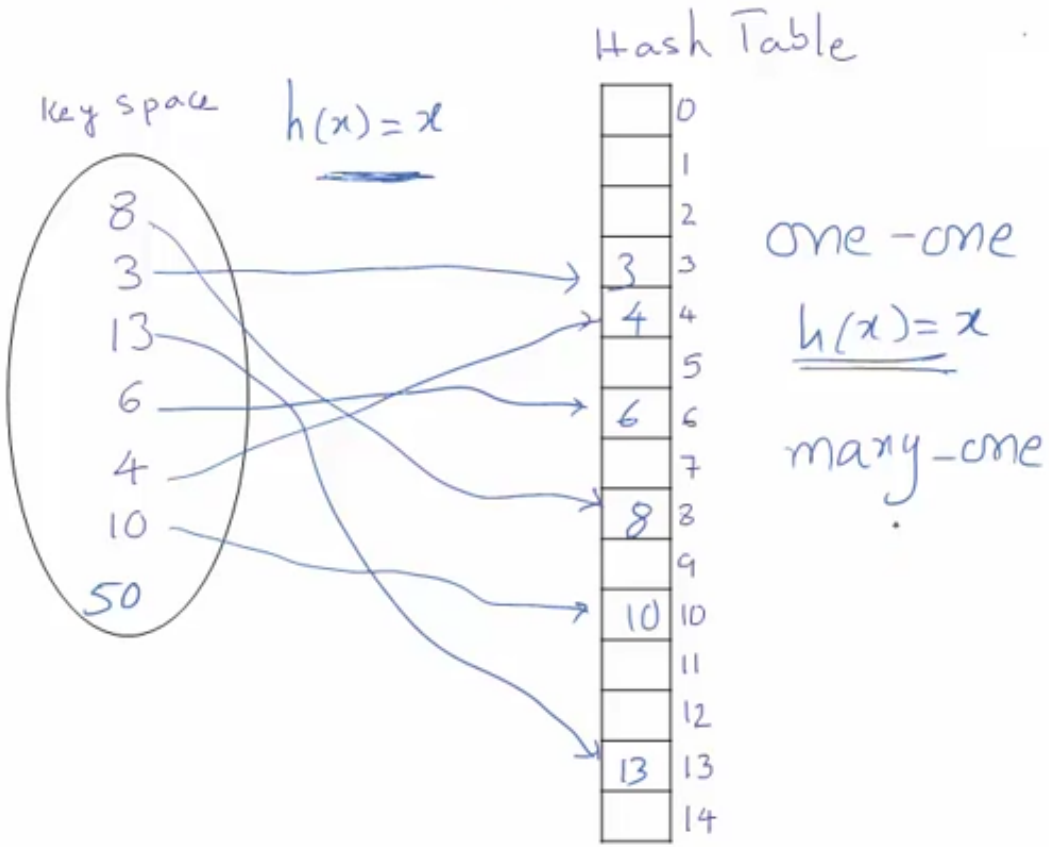


Hashing example

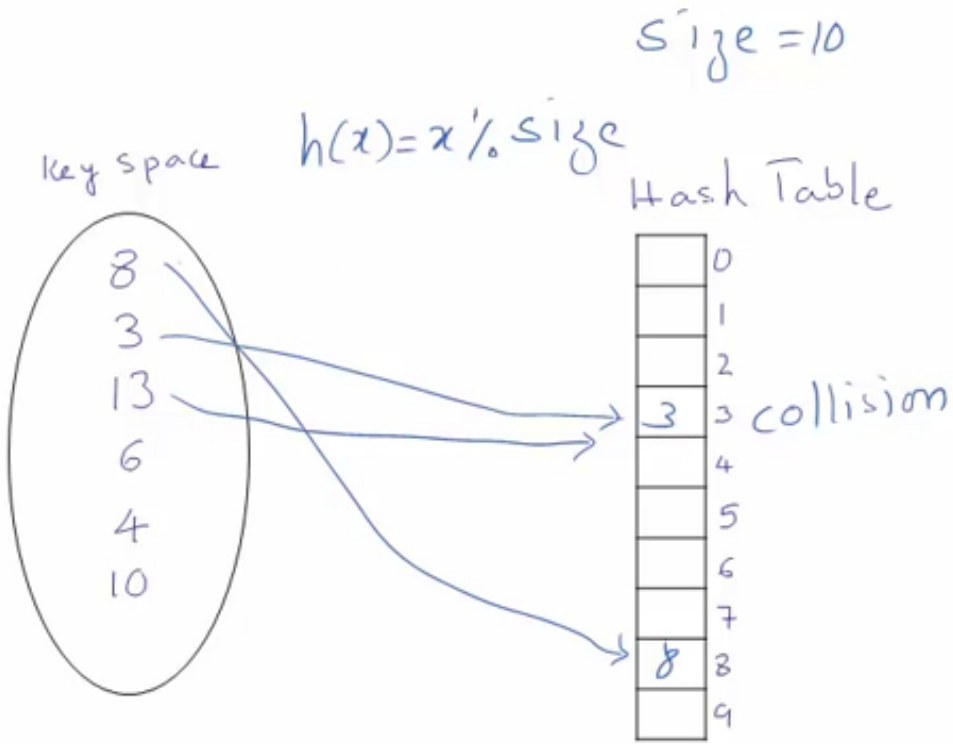


1. If we increase size of hash table we will still have O(1) of searching but a lot of space even there are planty of indices empty.
2. We use different mathematical model for hashing function.

Hashing example



Hashing example



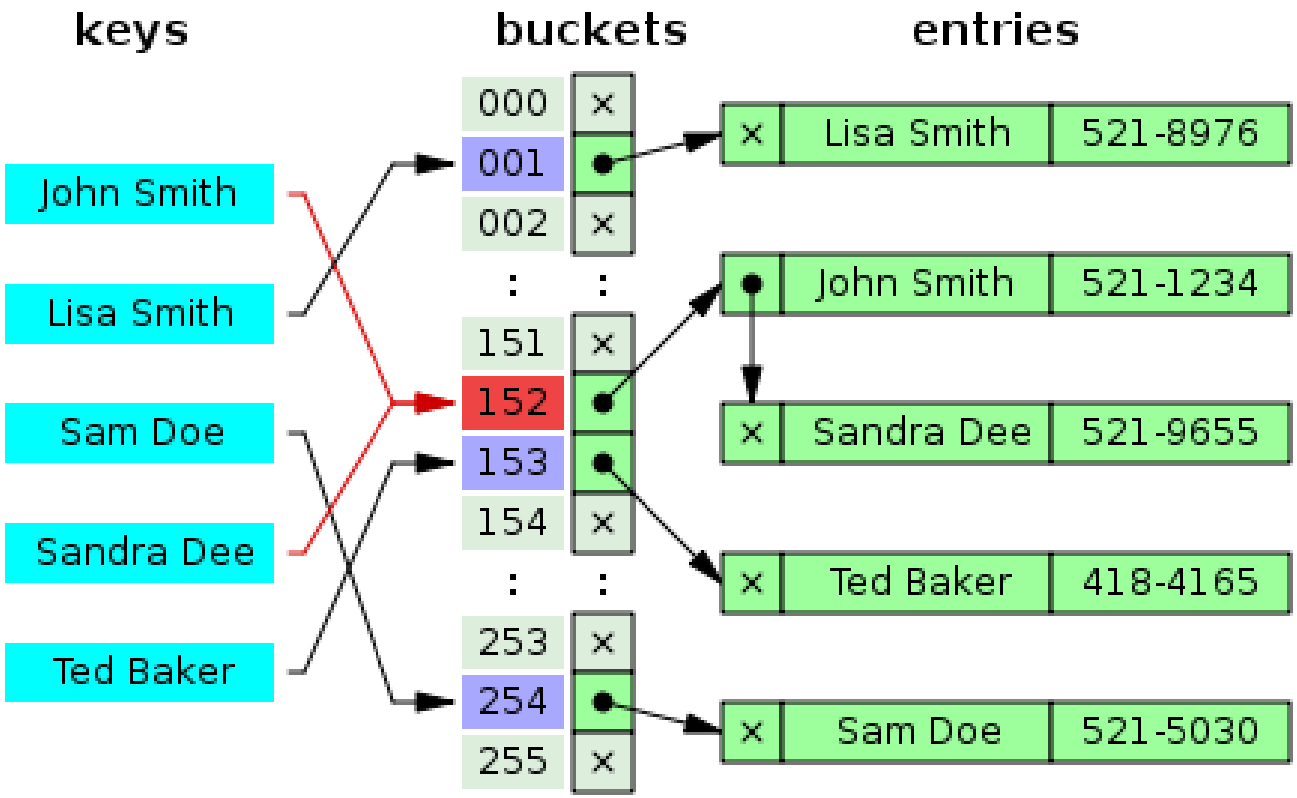
Collision

As we can see, it may happen that the hashing technique is used to create an already used index of the array. This called collision.

1. Separate chaining (Open hashing).
2. Open addressing (Close hashing).
   1. Linear probing.
   2. Quadratic probing.

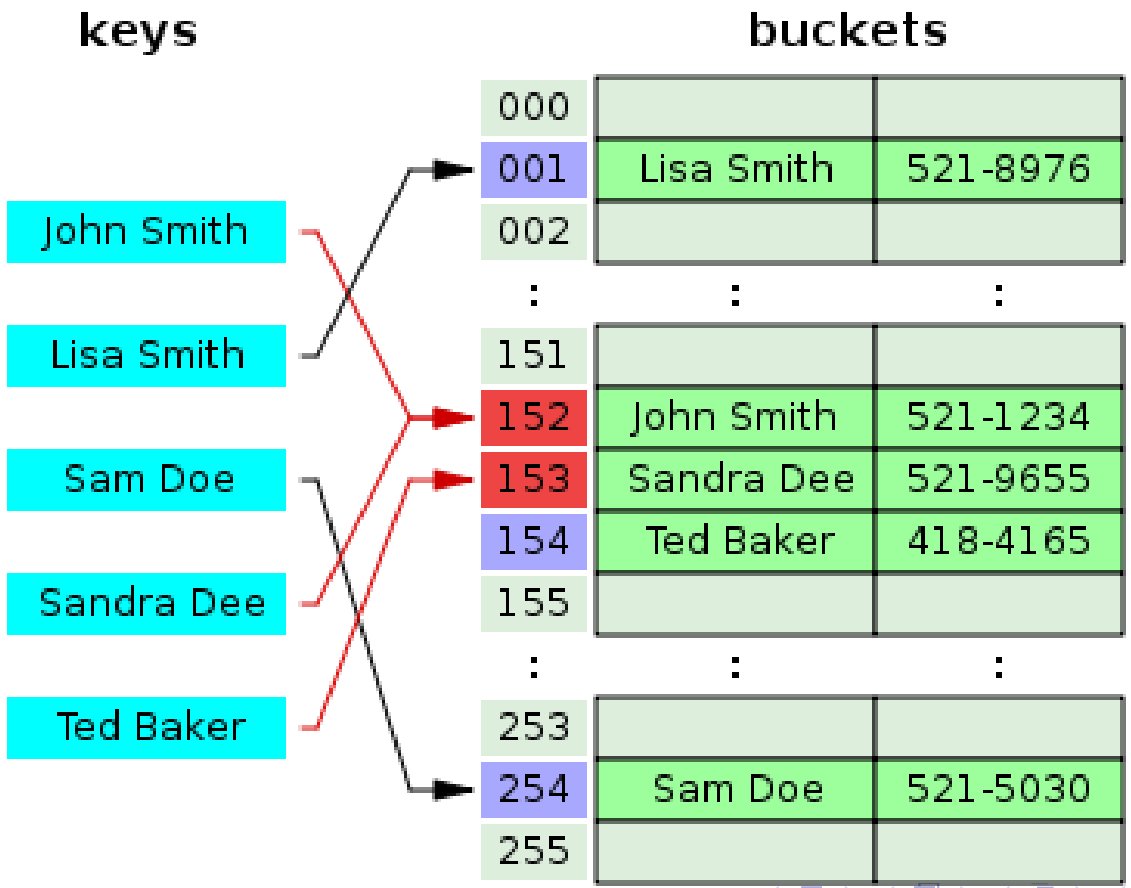
Collision

Collision resolution using separate chaining

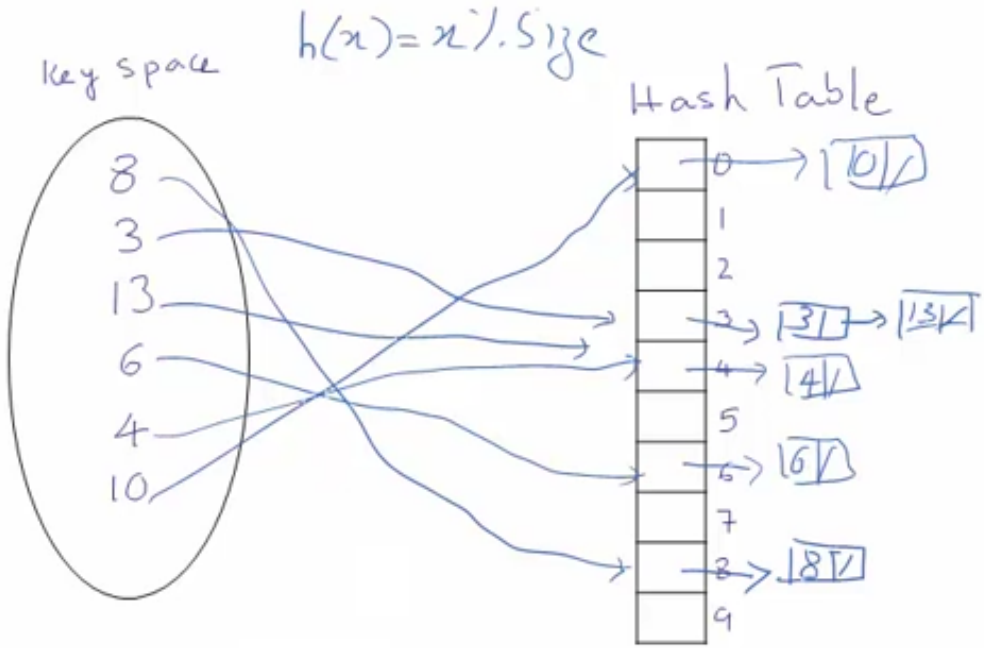


Collision

Collision resolution using open addressing

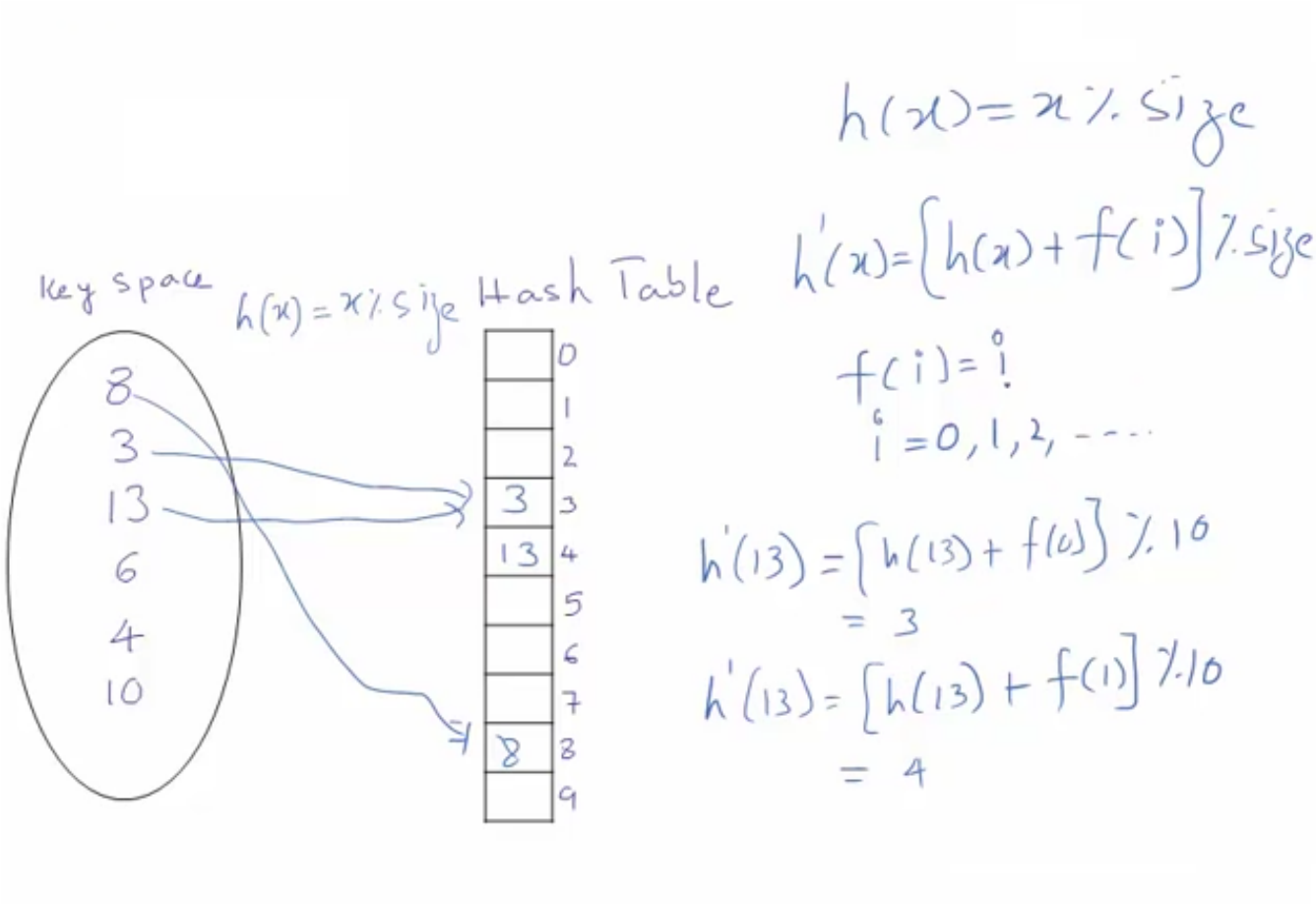


Hashing example using chaining resolution



Time complexity will be more than O(1) for searching. Why?

Hashing example using close hashing resolution



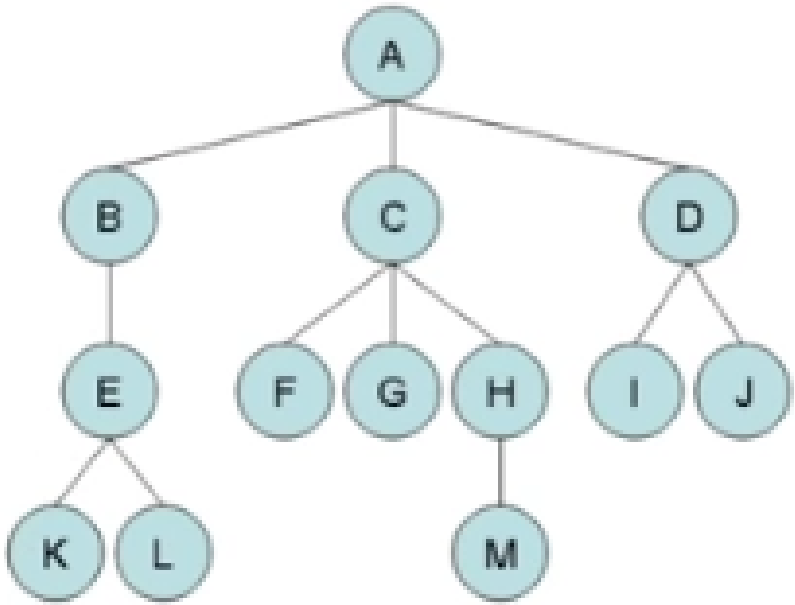
Time complexity will be more than O(1) for searching. Why?

Binary Tree

Tree

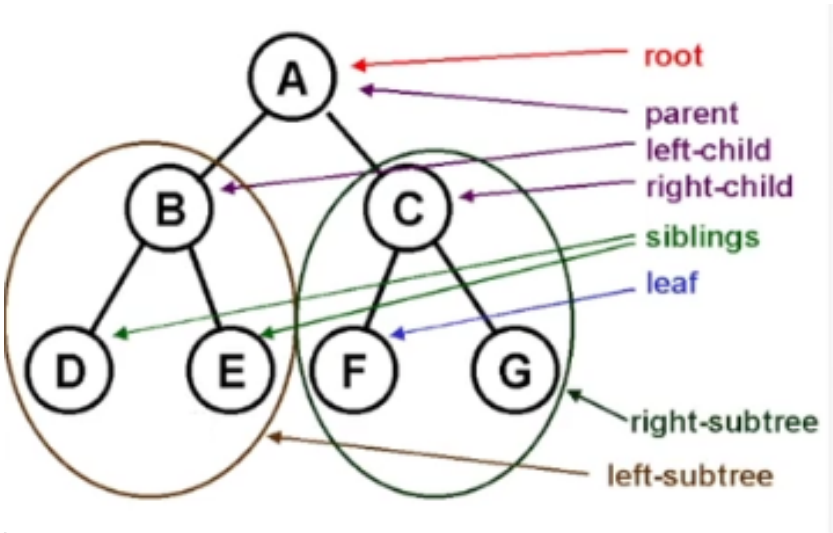
Binary Tree

Tree structure



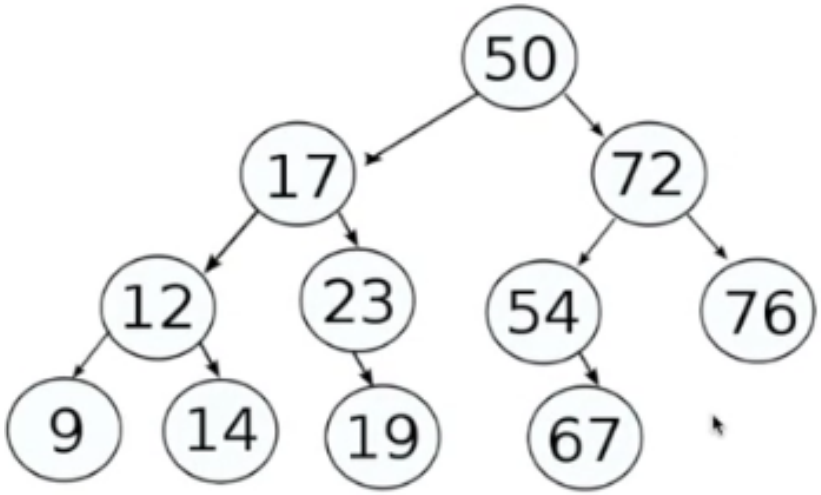
Binary Tree

Binary Tree structure



Binary Tree

Example of Binary Tree



A binary search tree adds these two characteristics:

- Each node has up to two children.
- For each node, its left descendents are less than the current node, which is less than the right descendents.

Binary Tree

Big O Notation of Binary Tree)

Algorithm	Average	Worst Case
Space	O(n)	O(n)
Search	**O(log n)**	**O(n)**
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Skip List

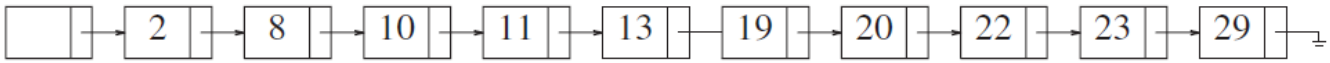


Figure Simple linked list

The idea is simple. starting with a simple linked list? what is the motivation:

1. We have an order linked list, but we cannot dive directly using BSearch.
2. The solution is put more layer above to the linkedList with a randomize appearance of the element (that's called randomization algorithms).

Skip List

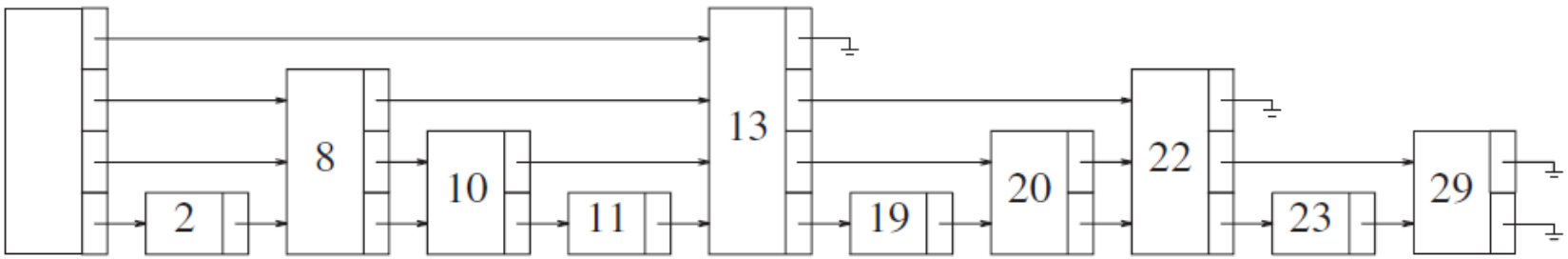


Figure A skip list

The linked list with four layer became a skiplis:

1. Each element have forward, backward, up, and down pointers.
2. Creating element of the other layer must be randomize.
3. This list will be behave like an ordered array with BSearch algorithm can be implemented.

Skip List

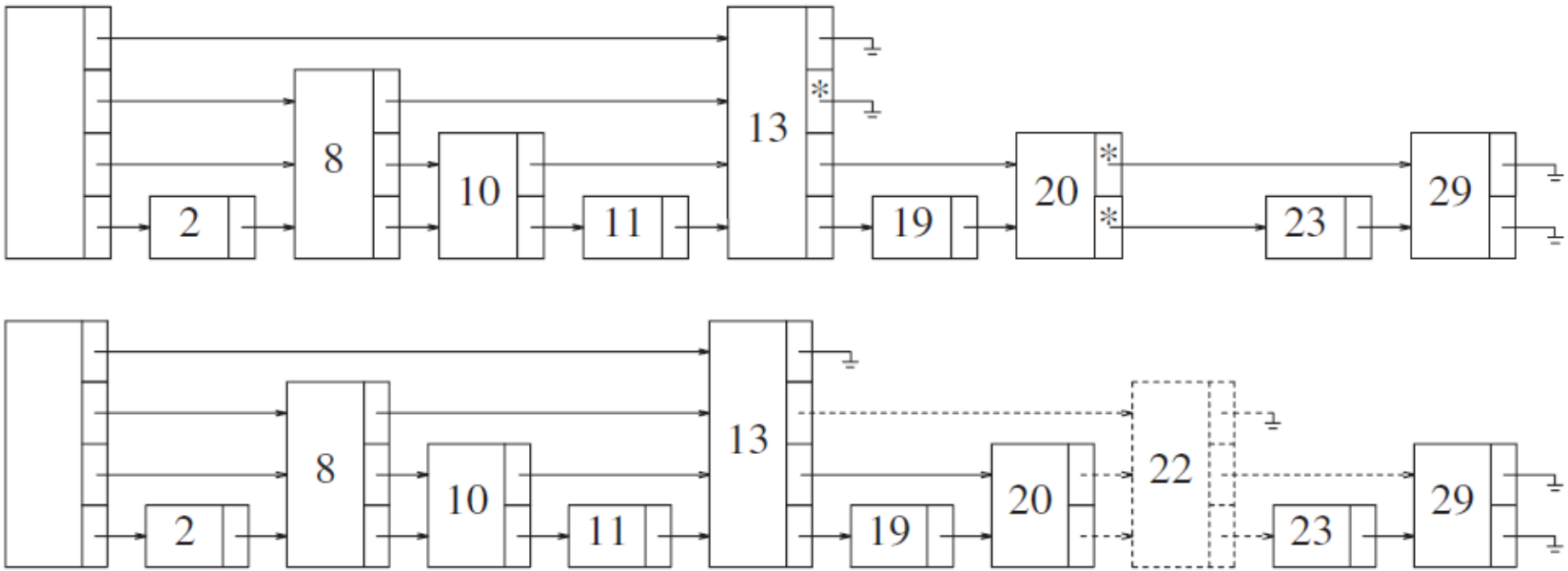


Figure Before and after an insertion

1. Start at the highest link at the header.
2. Traverse along this level until null or element larger than we search for.
3. When this occure we go to lower level and continue the strategy.

Skip List

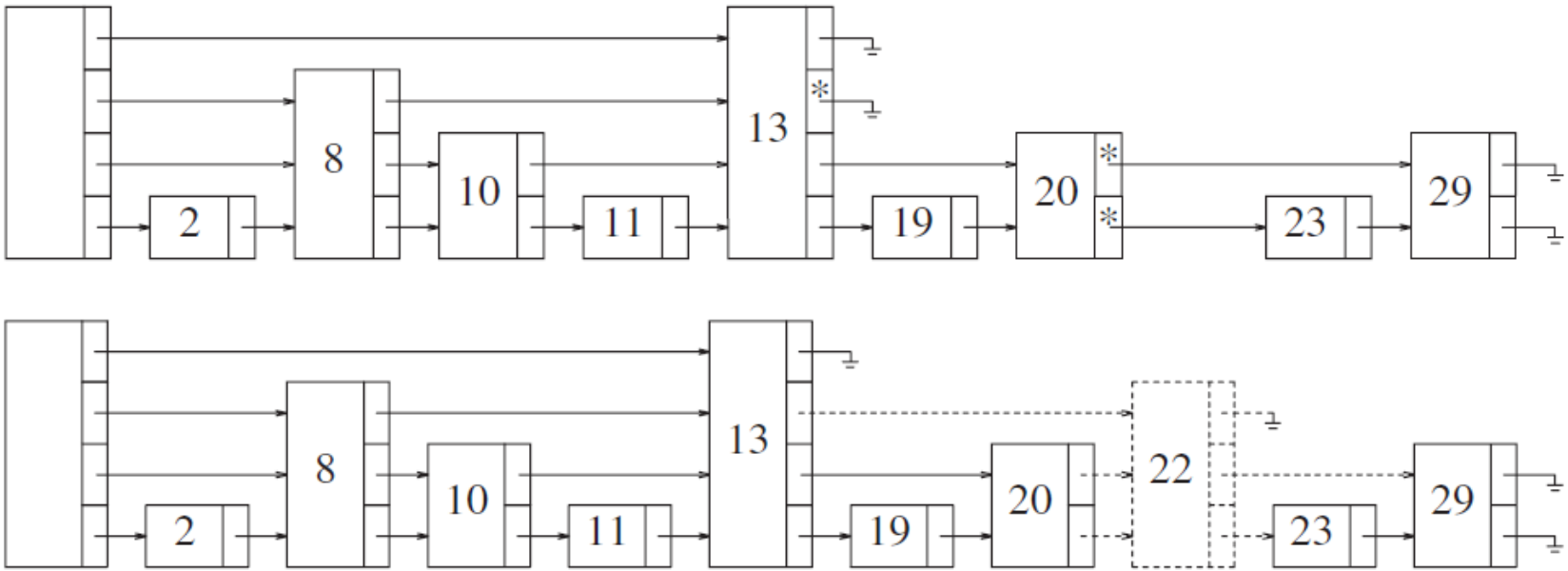


Figure Before and after an insertion

4. When reaching level one, either we are in the front of the element or is not in the list.
5. Insertion is like a search + we use randomization to find which layers the new element will appear.