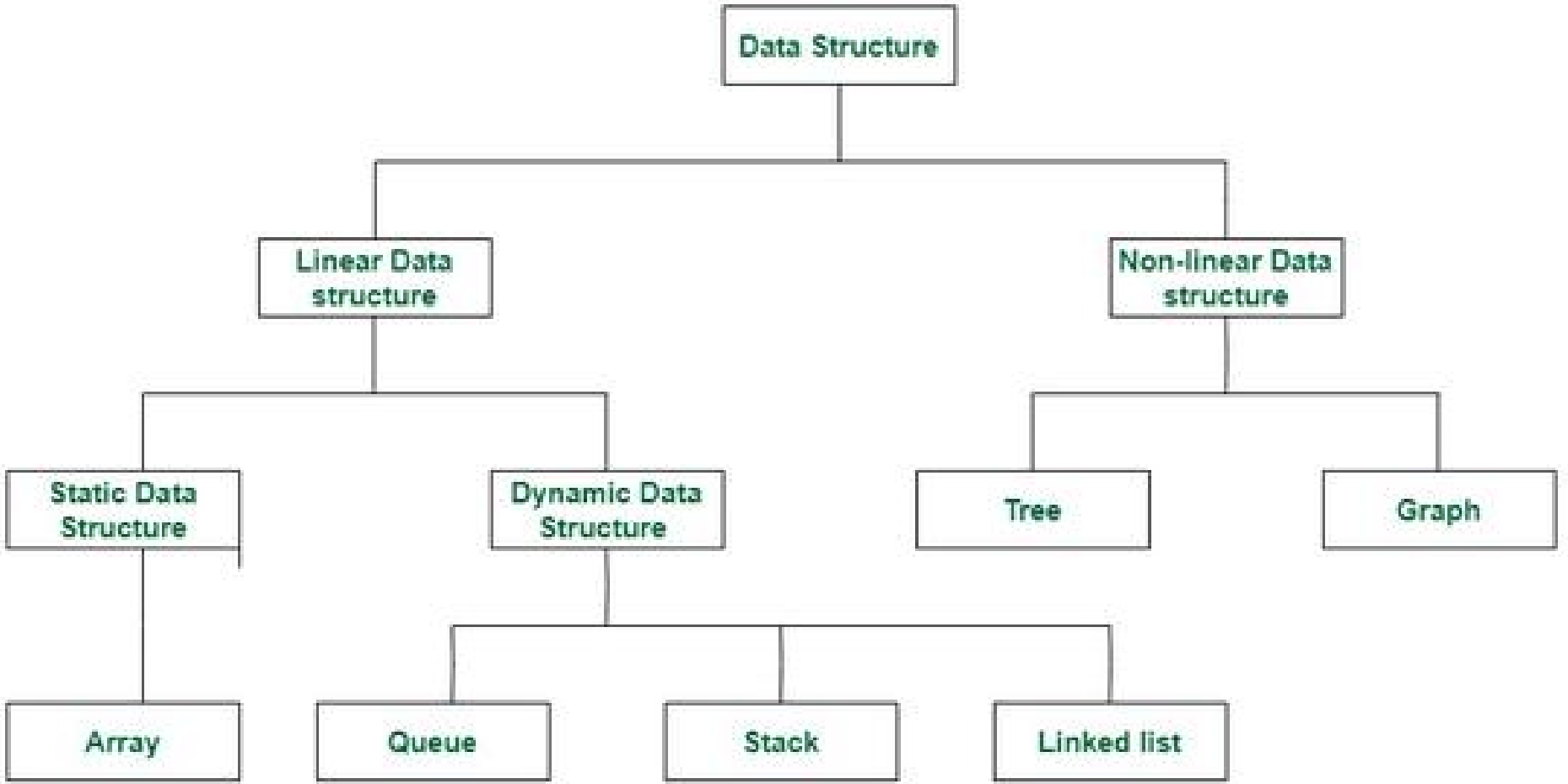
Data Structure and Algorithm

overview

data structure in many applications

Data type	Data structure
form of a variables to which a value can be assigned (same data type)	collection of different kinds of data
hold value	hold multiple types of data within a single object.
abstract implementation	concrete implementation.
no time complexity	time complexity plays an important role
only represents the type of data	the data and its value acquire the space in the computer’s main memory
Ex. int, float, double, etc.	Ex. stack, queue, tree, etc.

Classification of Data Structure



Linear data structure

- data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
- Examples : array, stack, queue, linked list, etc.

Static data structure

- has a fixed memory size.
- Example: array.

Dynamic data structure

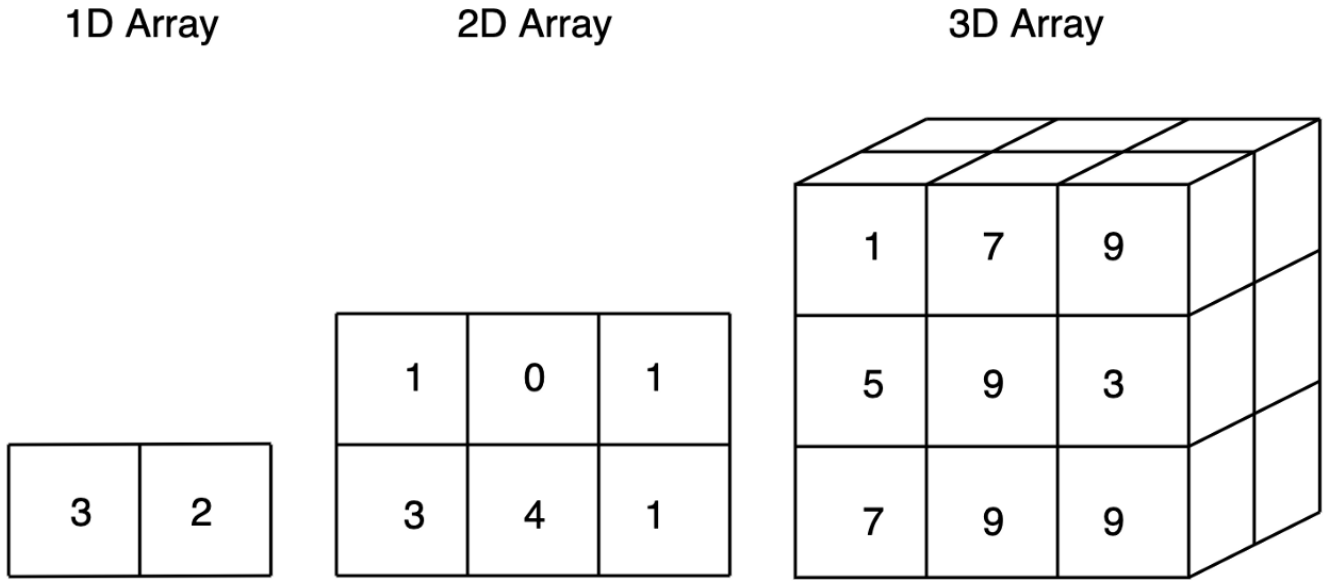
- can be randomly updated during the runtime
- efficient concerning the memory (space) complexity of the code.
- Examples of this data structure are queue, stack, etc.

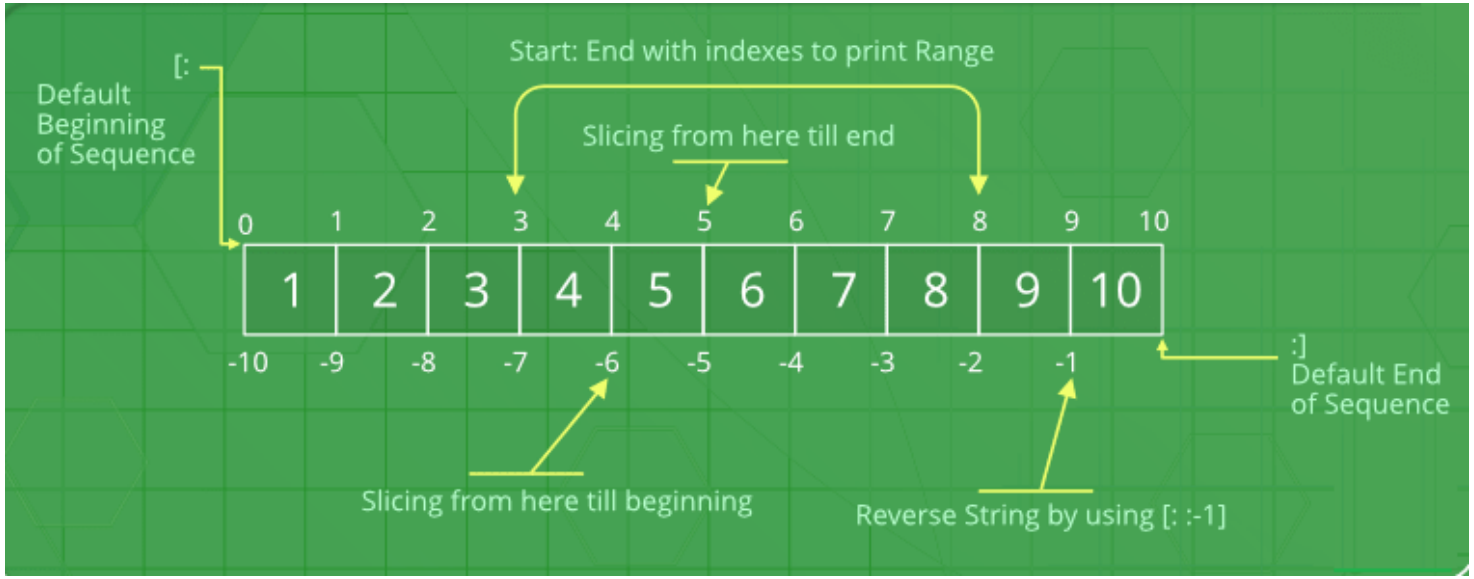
Non-linear data structure

- data elements are not placed sequentially or linearly are called non-linear data structures.
- we can’t traverse all the elements in a single run only.
- Examples of non-linear data structures are trees and graphs.

Array / List

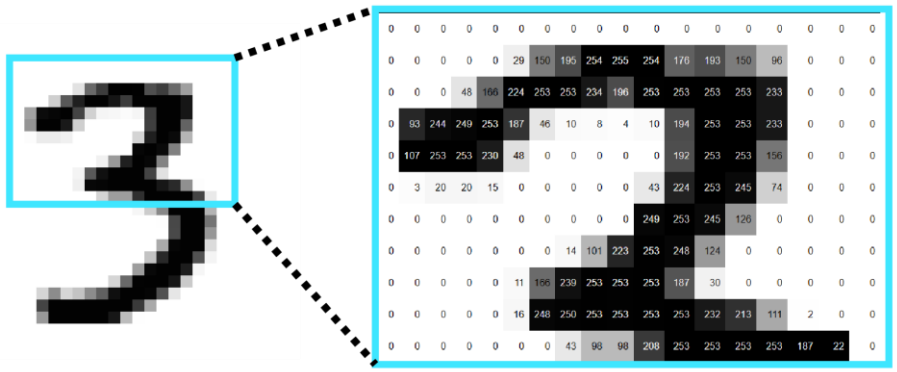
- collection of data items stored at contiguous memory locations
- store multiple items of the same type
- together adding an offset to a base value





Applications of Array:

- image processing.



- solving matrix problems.

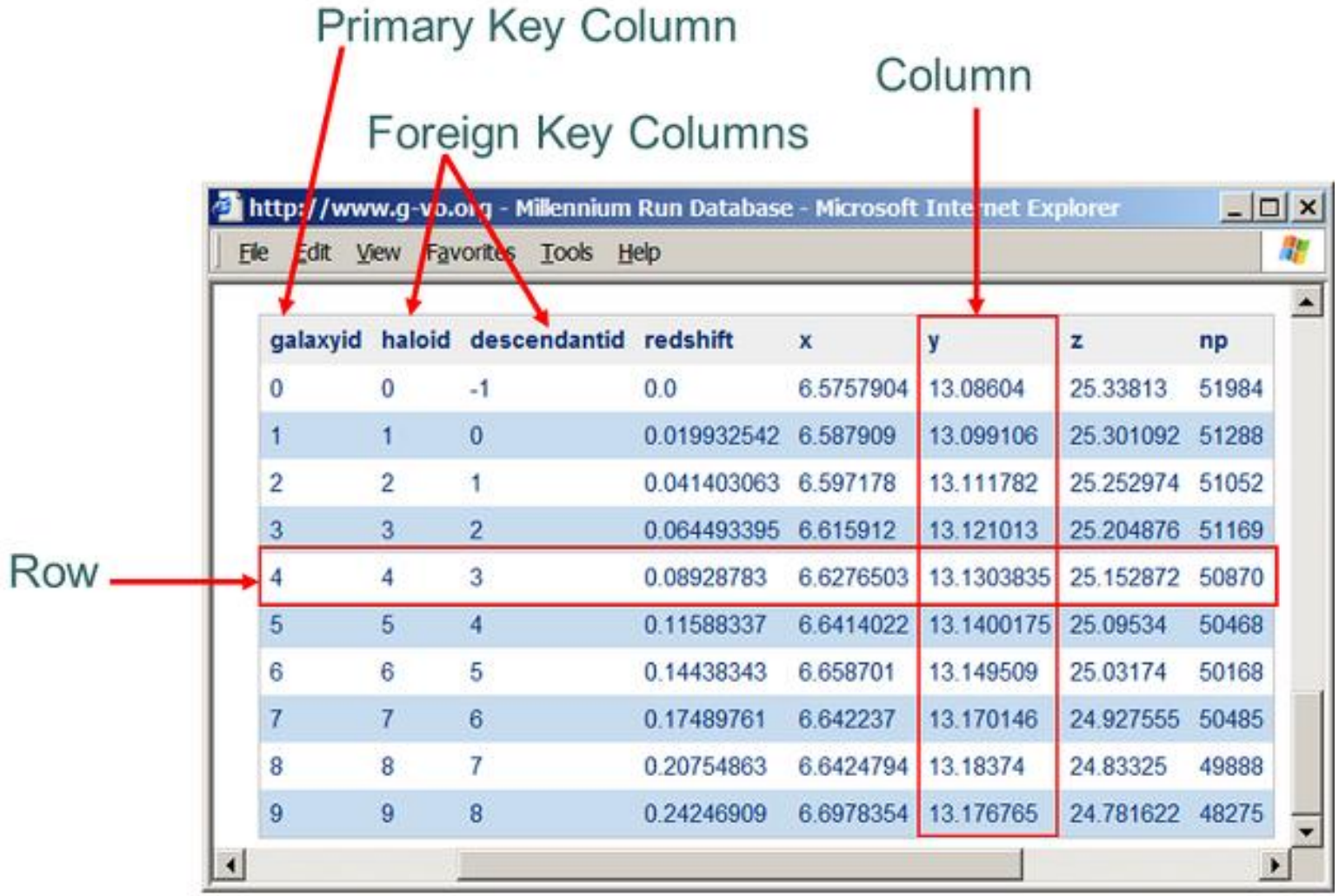
$$\mathbf{A} + \mathbf{B} = \mathbf{C}$$

$$\begin{bmatrix} 2 & 1 \\ 3 & 2 \\ -2 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 4 & 2 \\ -2 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 7 & 4 \\ -4 & 3 \end{bmatrix}$$

$$(3,2) \quad (3,2) \quad (3,2)$$

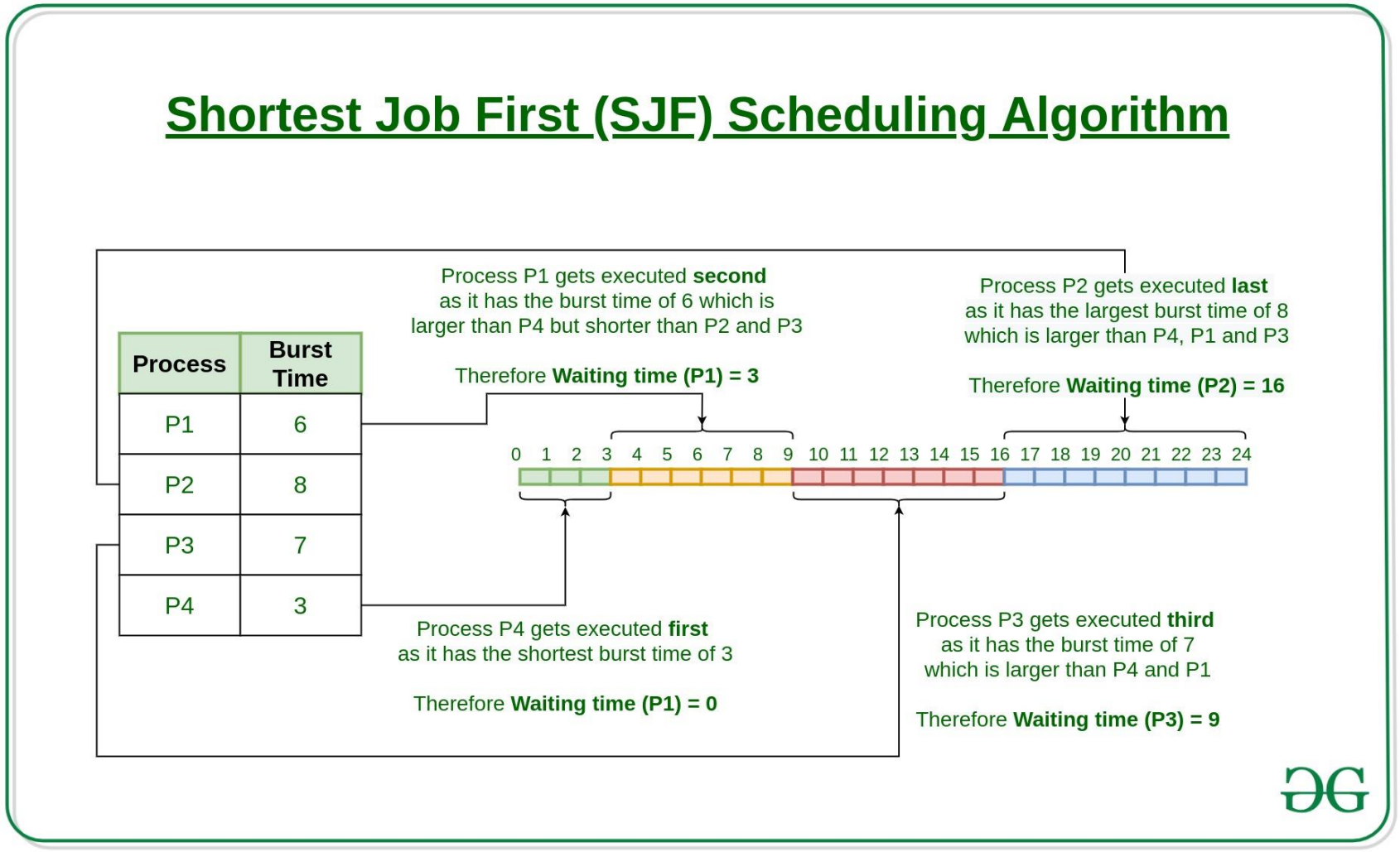
Applications of Array:

- Database records are also implemented by an array.



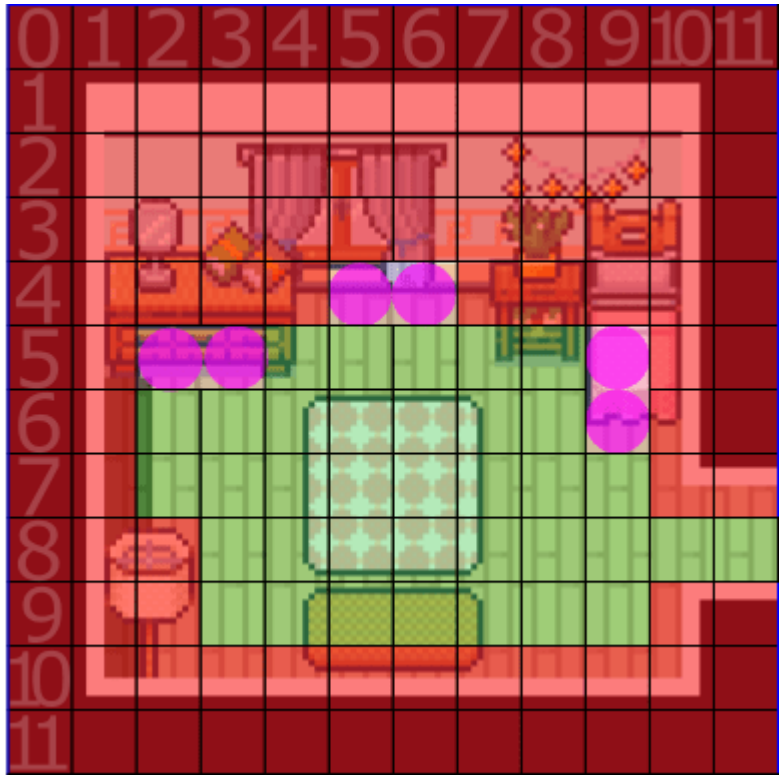
Applications of Array:

- CPU scheduling.



Applications of Array:

- Game map array




```
const X = { walk: false, action: false,};
const O = { walk: true, action: false,};
const AO = { walk: true, action: true,};
const AX = { walk: false, action: true,};

const BEDROOM_MAP = [ //each elem in the nested array equals a tile on
    [X, X, X, X, X, X, X, X, X, X, X, X], // y = 0
    [X, X, X, X, X, X, X, X, X, X, X, X], // y = 1
    [X, X, X, X, X, X, X, X, X, X, X, X], // y= 2
    [X, X, X, X, X, X, X, X, X, X, X, X], // y = 3
    [X, X, AX, AX, X, AO, AO, X, AO, AO, X, X], // y = 4
    [X, X, AO, AO, O, O, O, O, AO, AO, X, X], // y = 5
    [X, X, O, O, O, O, O, O, O, O, X, X], // y = 6
    [X, X, O, O, O, O, O, O, O, O, O, O], // y = 7
    [X, X, X, O, O, O, O, O, O, O, O, O], // y = 8
    [X, X, X, O, O, O, O, O, O, O, X, X], // y = 9
    [X, X, X, X, X, X, X, X, X, X, X, X], // y = 10
    [X, X, X, X, X, X, X, X, X, X, X, X], // y = 11
]
```

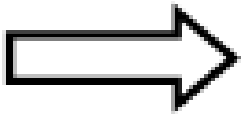
Applications of Array:

- Sparse Matrix : A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. **If most of the elements of the matrix have 0 value, then it is called a sparse matrix.**

- Why to use Sparse Matrix instead of simple matrix ?
  - Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
  - Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Sparse Matrix using array

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$



Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Applications of Array:

- It helps in implementing a sorting algorithm

Applications of Array:

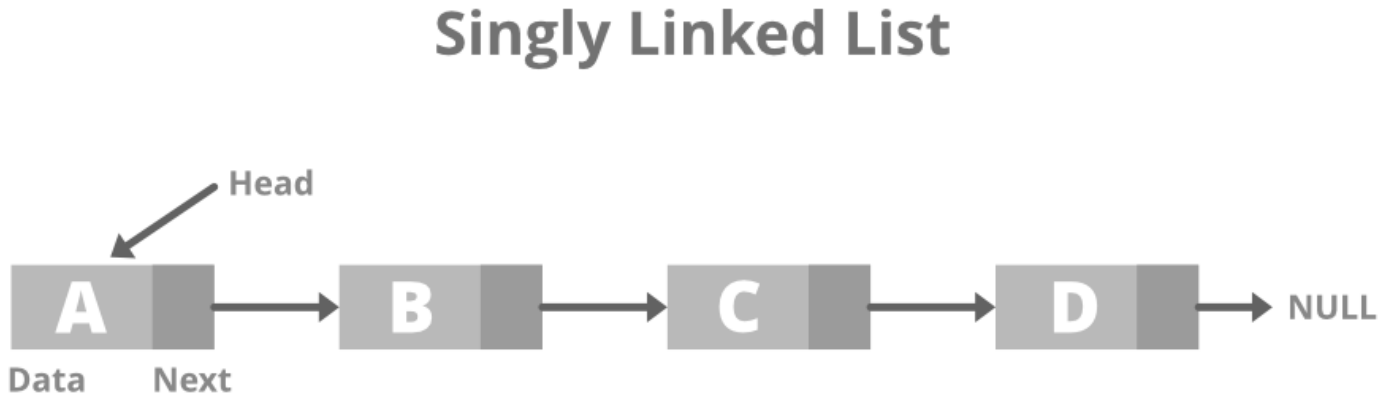
- implement other data structures like Stacks, Queues, Heaps, Hash tables, etc.
- a lookup table in computers.
- speech processing
- Etc.

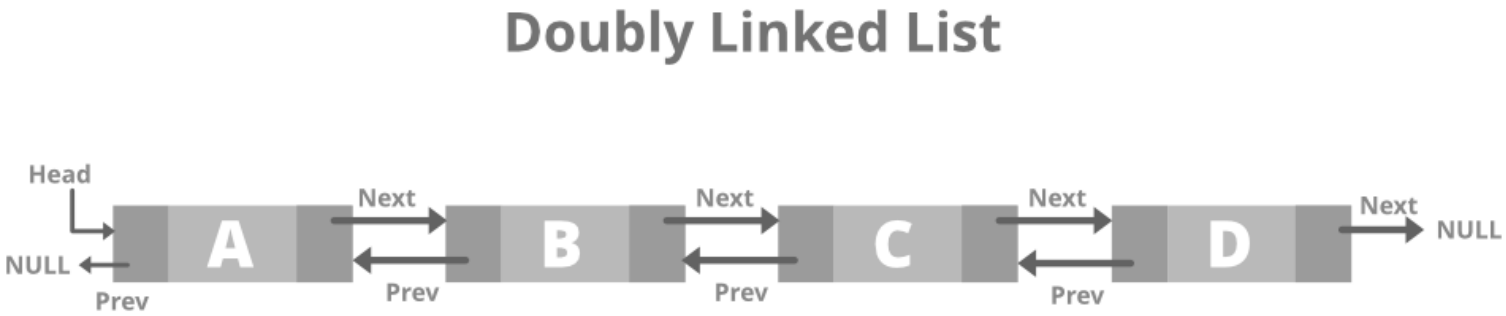
```
my_list = ['p', 'r', 'o', 'b', 'e']

# first item
print(my_list[0])  # p
# third item
print(my_list[2])  # o
# fifth item
print(my_list[4])  # e

# Nested List
n_list = ["Happy", [2, 0, 1, 5]]
# Nested indexing
print(n_list[0][1])
print(n_list[1][3])
# Error! Only integer can be used for indexing
print(my_list[4.0])
```

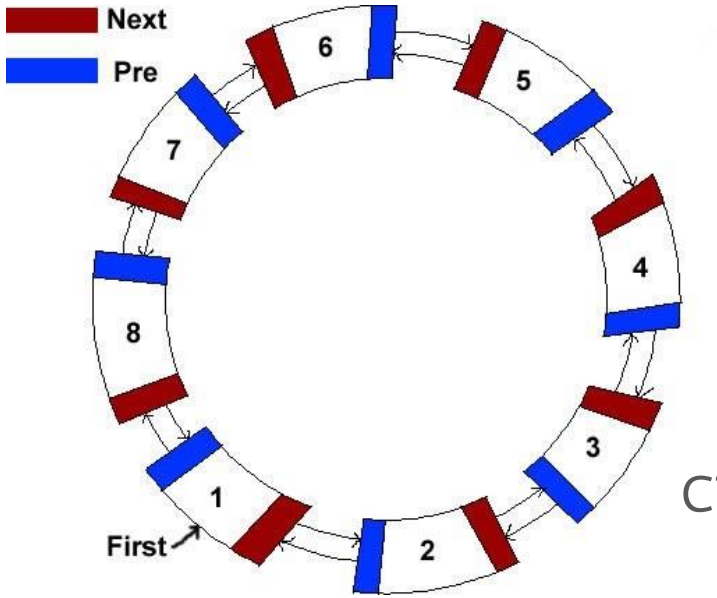
```
---------- : Output : ----------

p
o
e
a
5
Traceback (most recent call last):
  File "<string>", line 21, in <module>
TypeError: list indices must be integers or slices, not
float
```

Linked List

- Linked list is a linear data structure.
- linked list elements are not stored at a contiguous location
- the elements are linked using pointers.







Circular Doubly Linked List

Types of linked list

Singly Linked List

- every node stores address or reference of the next node in the list
- the last node has the next address or reference as NULL.
- For example 1->2->3->4->NULL

Doubly Linked List

- there are two references associated with each node
- One of the reference points to the next node and one to the previous node.
- can traverse in both directions and for deletion,
- Eg. NULL<-1<->2<->3->NULL

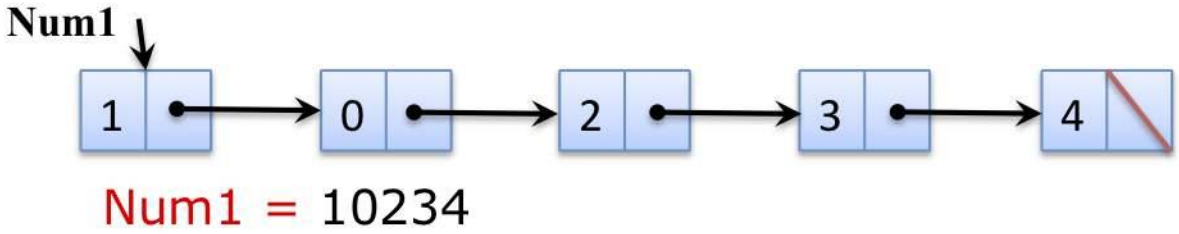
Circular Linked List

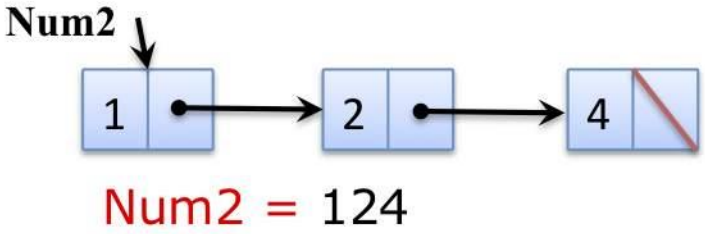
- a linked list where all nodes are connected to form a circle , no NULL at the end.
- can be a singly circular linked list or a doubly circular linked list.
- any node can be made as starting node.
- Eg. 1->2->3->1 [The next pointer of the last node is pointing to the first]

Circular Doubly Linked List

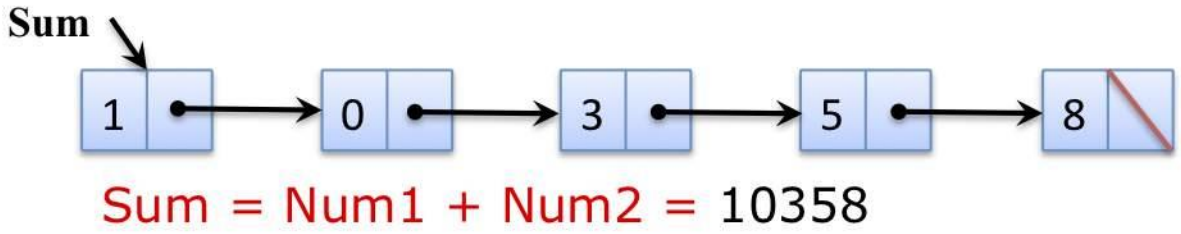
- a combination of the doubly linked list and the circular linked list.
- bidirectional and contains two pointers and the last pointer points to the first pointer.

Applications of linked list:





- Adding super long integer



Applications of linked list:

- Adding two polynomial

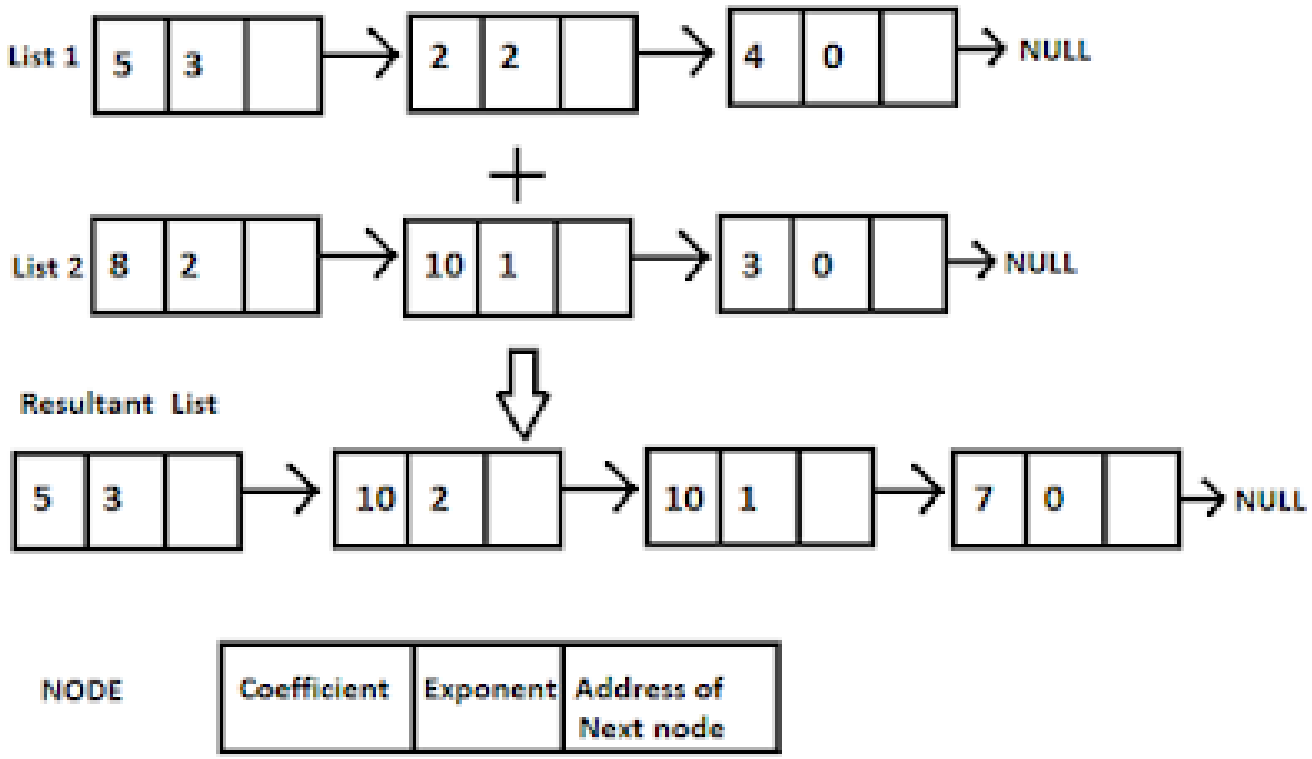
$5x^3 + 2x^2 + 4$

+

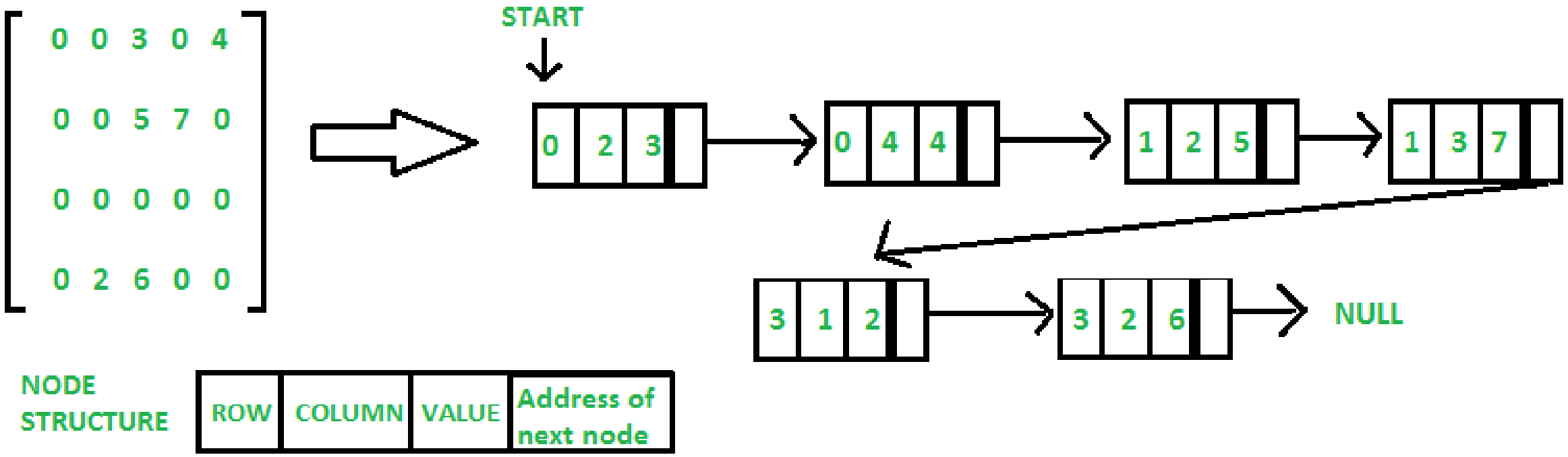
$8x^2 + 10x + 3$

=

$5x^3 + 10x^2 + 10x + 7$

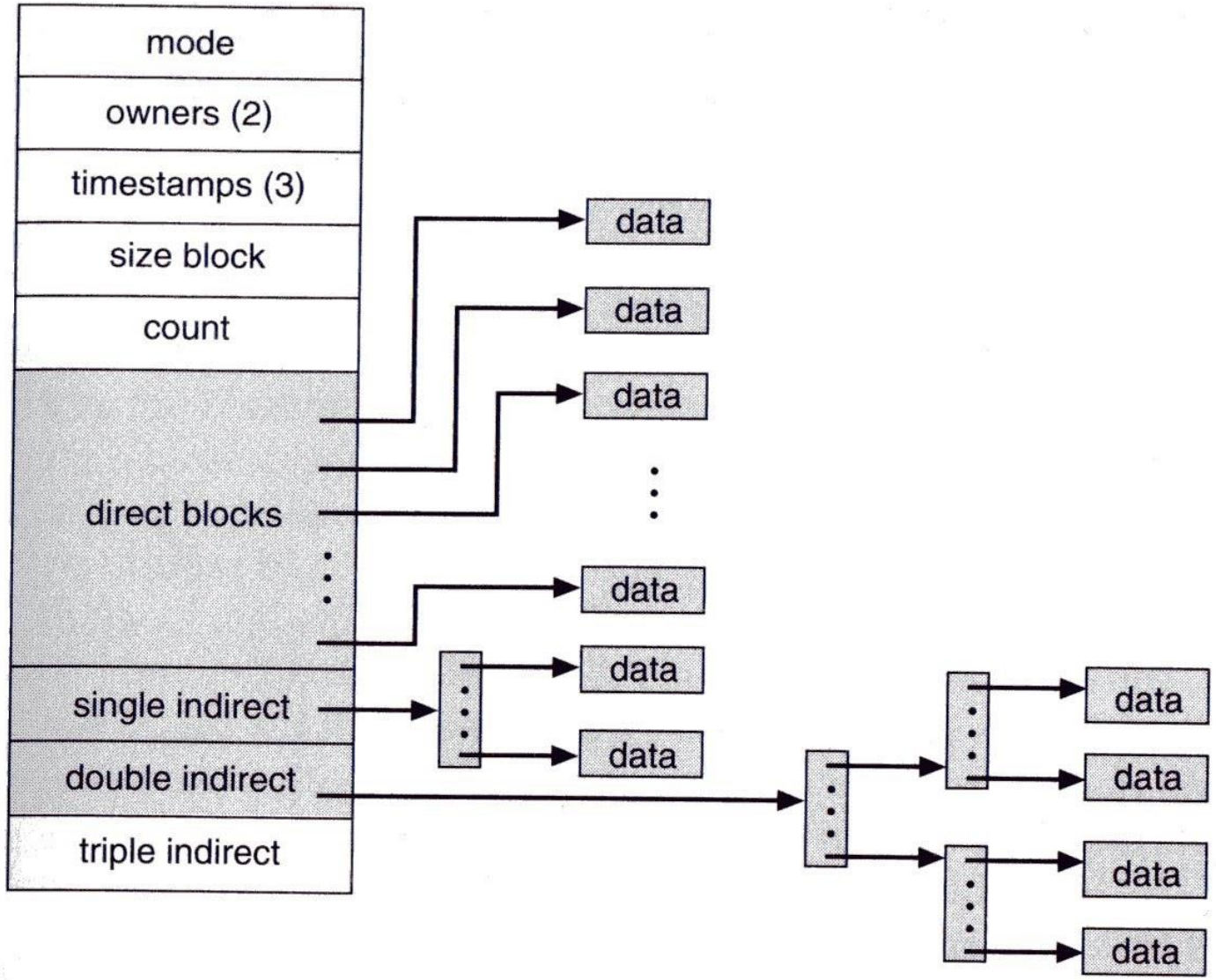


Sparse Matrix using link list



Unix i-node

- Every file in a UNIX file system is uniquely represented by an i-node.
- Each i-node has a unique i-node number as its identifier.



Memory management using link list

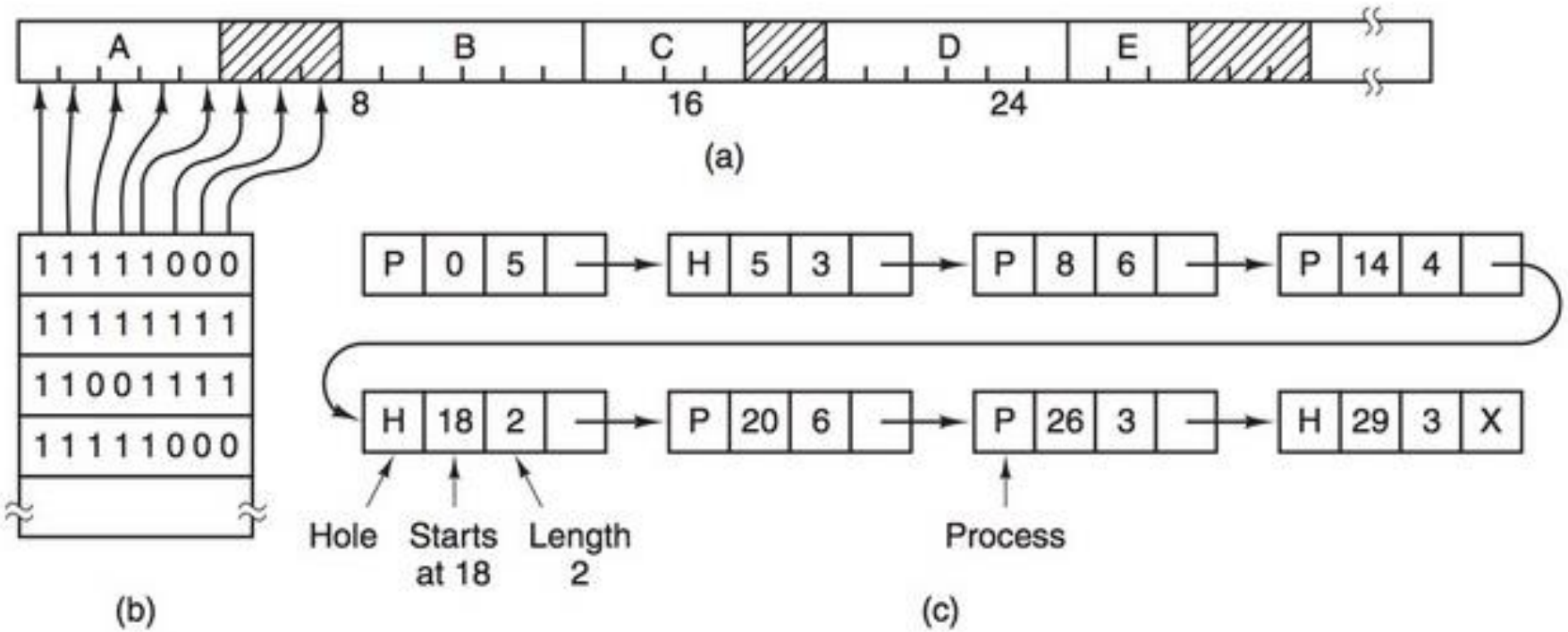


Figure 3-6. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

Round robin in multi-player game:

Implement by linked list

Applications of linked list:

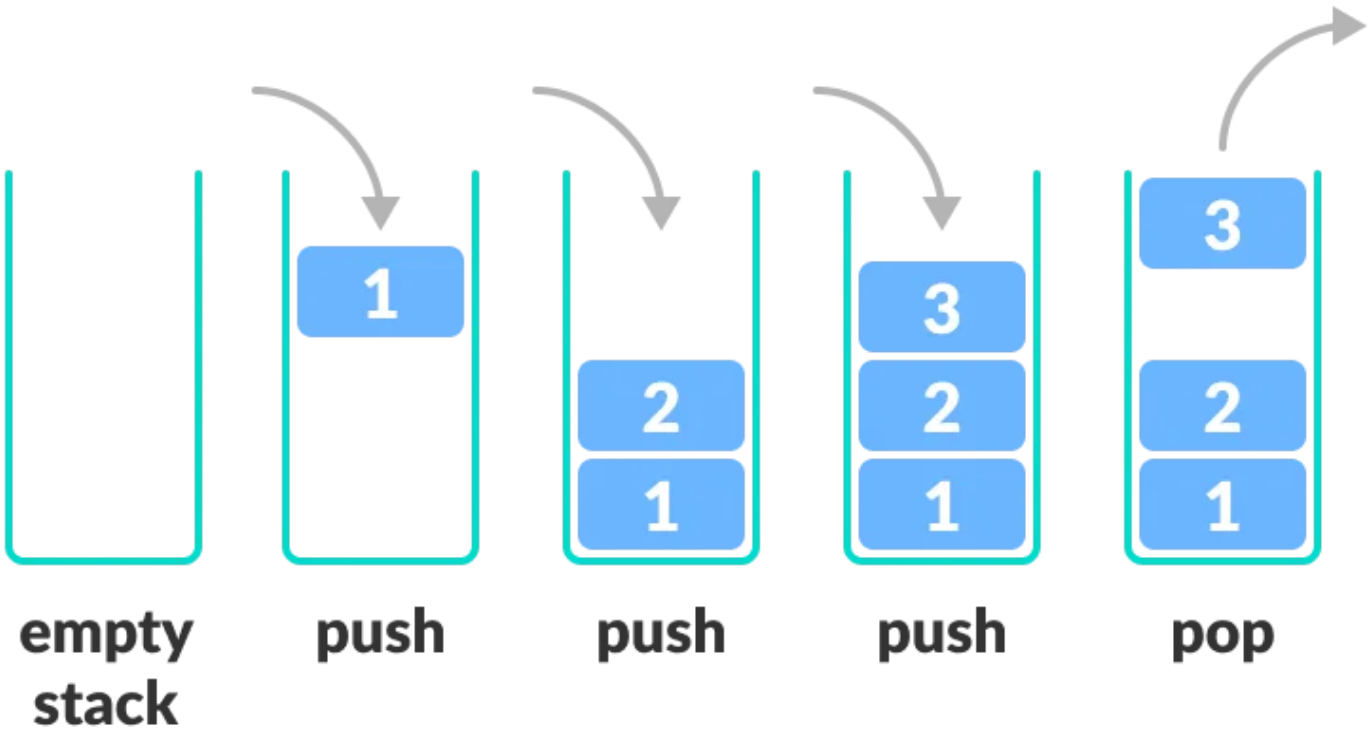
- Bi-directional linked list is used in image viewer. The previous and next images are linked, hence can be accessed by the previous and next buttons.
- In a music playlist, songs are linked to the previous and next songs.
- Linked lists are used to implement stacks, queues, graphs, etc.

```
class Node:
    def __init__(self, dataval=None):
        self.dataval = dataval
        self.nextval = None

class SLinkedList:
    def __init__(self):
        self.headval = None

    def listprint(self):
        printval = self.headval
        while printval is not None:
            print (printval.dataval)
            printval = printval.nextval

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

# Link first Node to second node
list.headval.nextval = e2

# Link second Node to third node
e2.nextval = e3

list.listprint()
```

---------- : Output : ----------

```
Mon
Tue
Wed
```

Stack



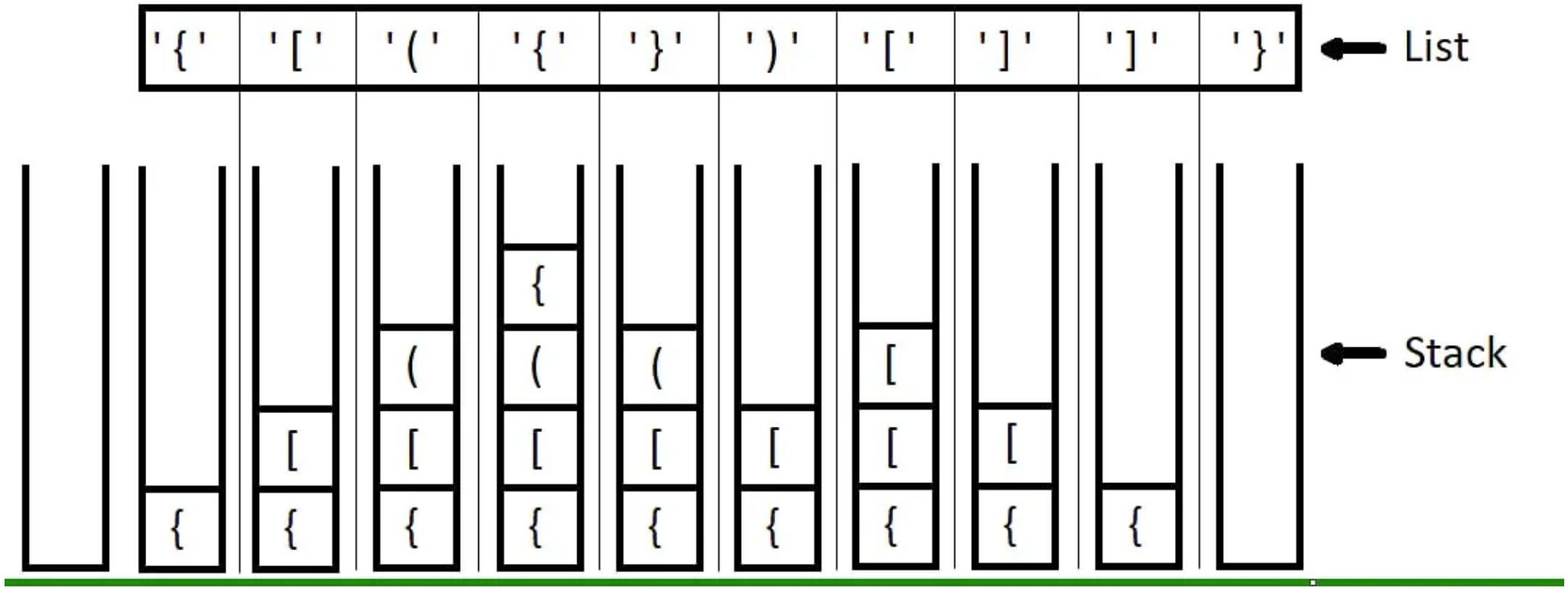
- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The order may be LIFO (Last In First Out) or FILO (First In Last Out).
- In stack, all insertion and deletion are permitted at only one end of the list.

three basic operations are performed in the stack:

- Initialize: Make a stack empty.
- Push: Adds an item in the stack.
  - If the stack is full, then it is said to be an Overflow condition.
- Pop: Removes an item from the stack.
  - The items are popped in the reversed order in which they are pushed.
  - If the stack is empty, then it is said to be an Underflow condition.
- Peek or Top: Returns top element of the stack.
- isEmpty: Returns true if the stack is empty, else false.

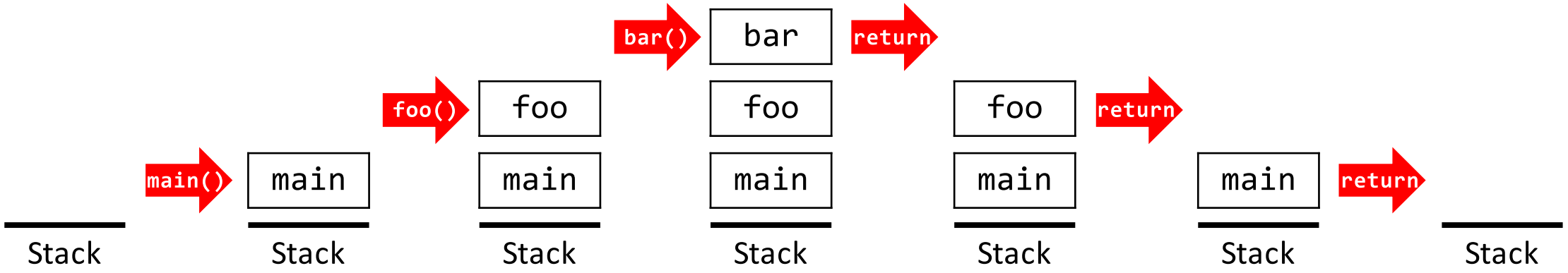
Applications of stack:

- parenthesis checking.



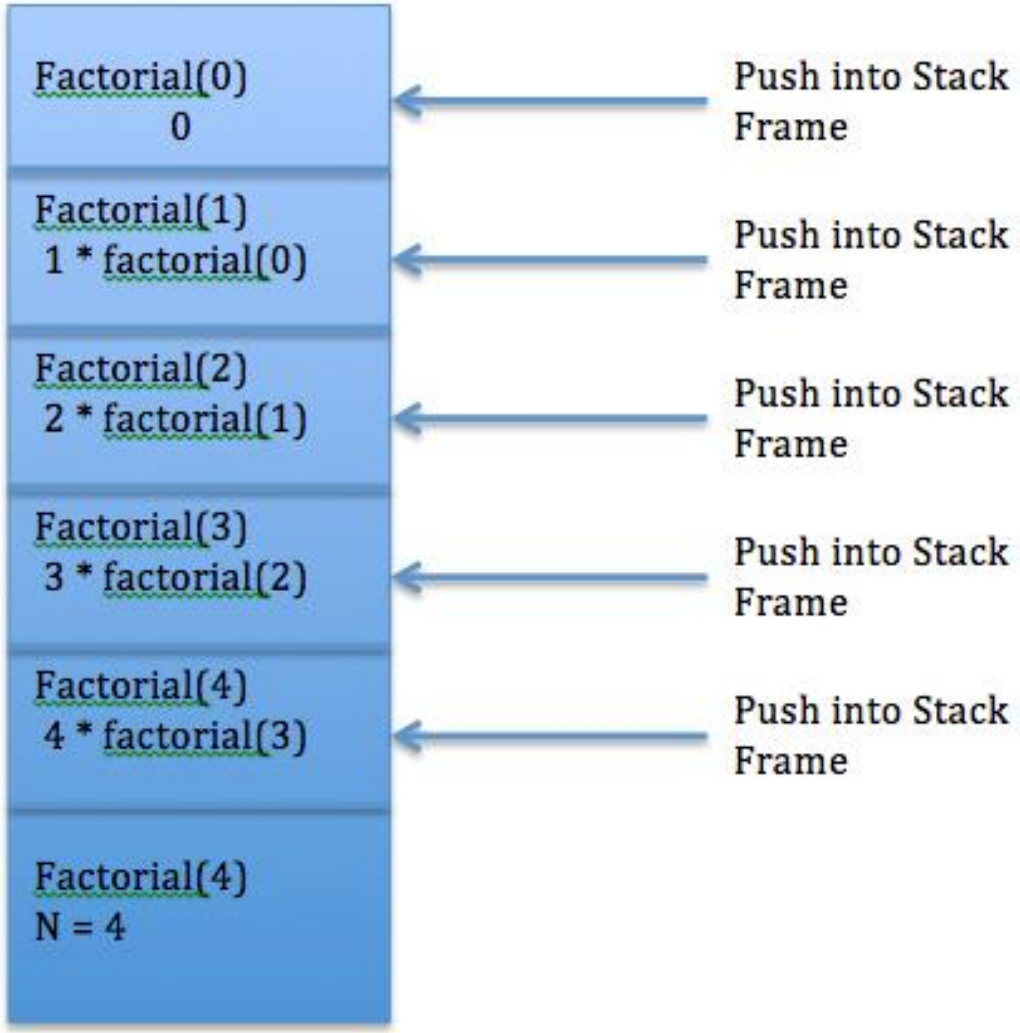
Applications of stack:

- processing of function calls



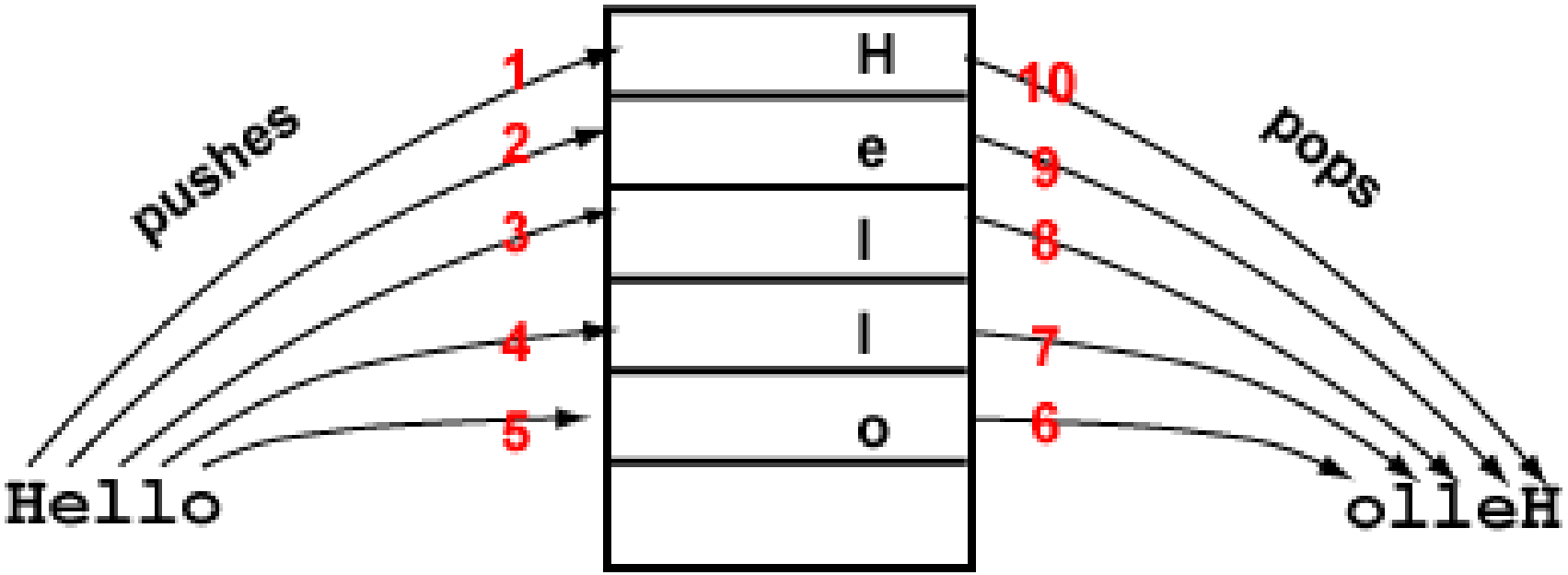
Applications of stack:

- Implement recursion method



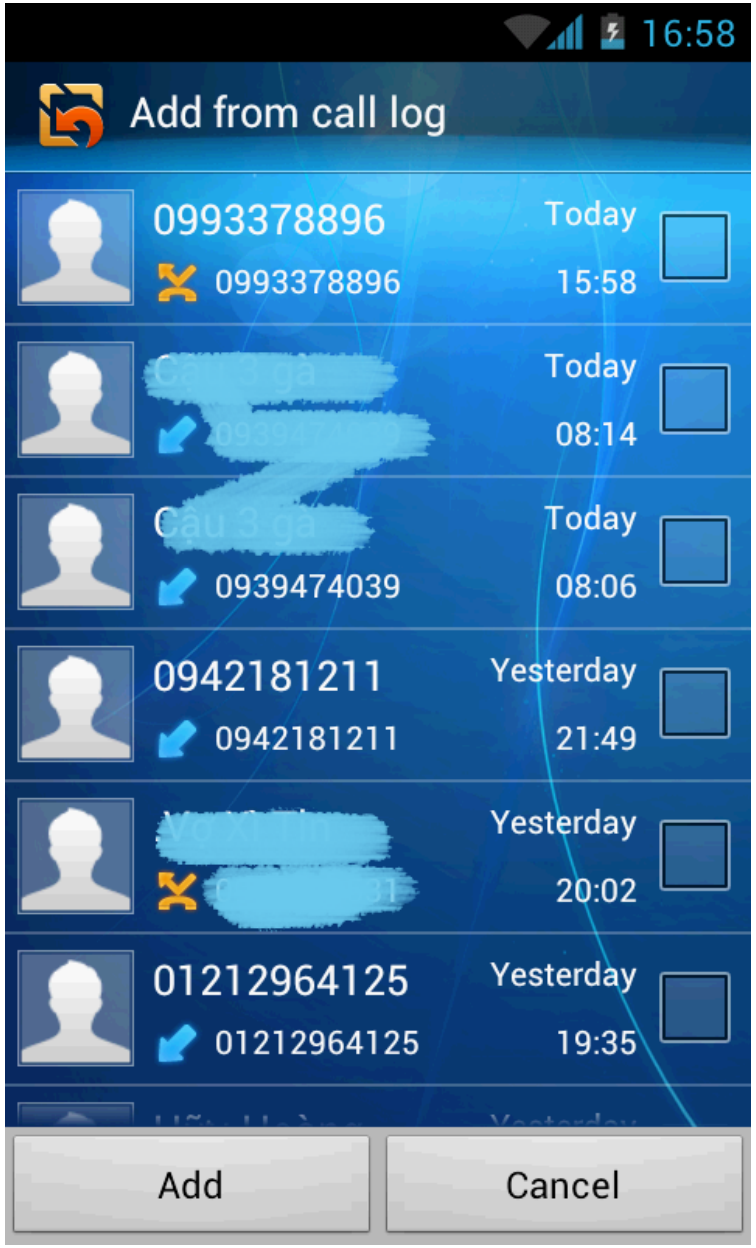
Applications of stack:

- reversing a string



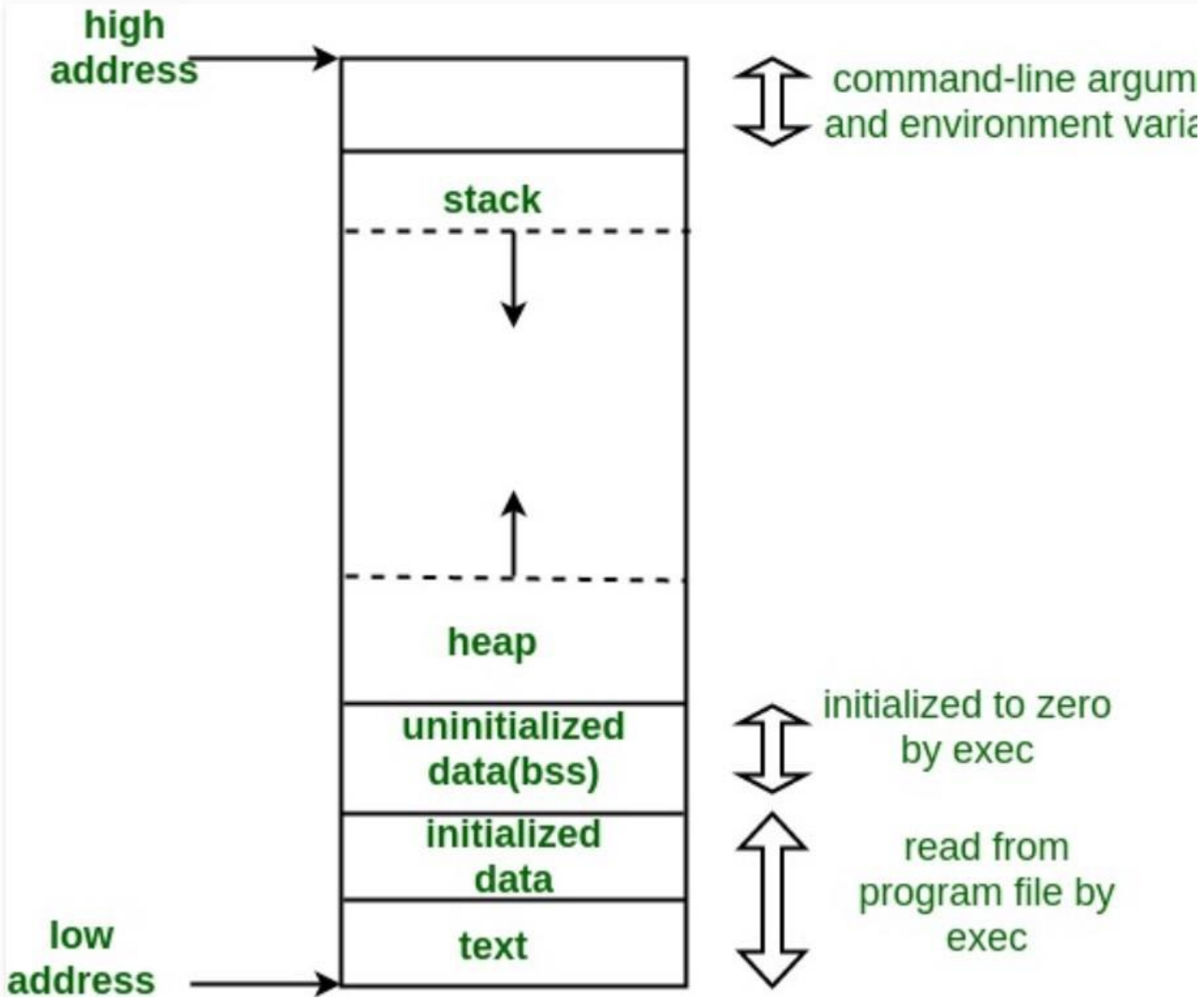
Applications of linked list:

- Call log in mobile also uses stack data structure.



Applications of linked list:

- Stack is used in memory management.
- The stack is a segment of memory where data like your local variables and function calls get added and/or removed in a last-in-first-out (LIFO) manner.



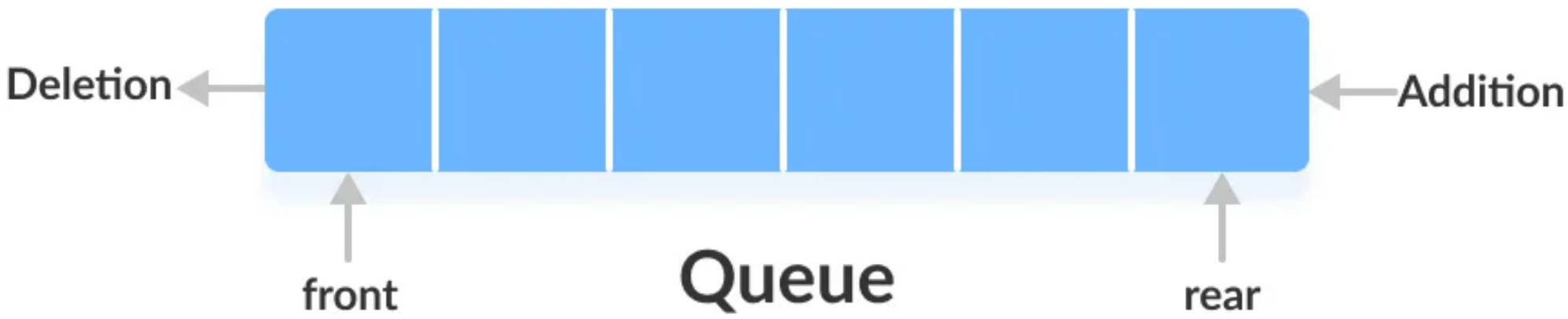
```
# Python program to
# demonstrate stack implementation
# using list

stack = []

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack')
print(stack)

# pop() function to pop
# element from stack in
# LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are popped:')
print(stack)

# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

---------- : Output : ----------

```
Initial stack
['a', 'b', 'c']

Elements popped from stack:
c
b
a

Stack after elements are popped:
[]
```

Queue

- Linear structure
- First In First Out (FIFO),inserted at one end and deleted from the other end



Queue : four basic operations

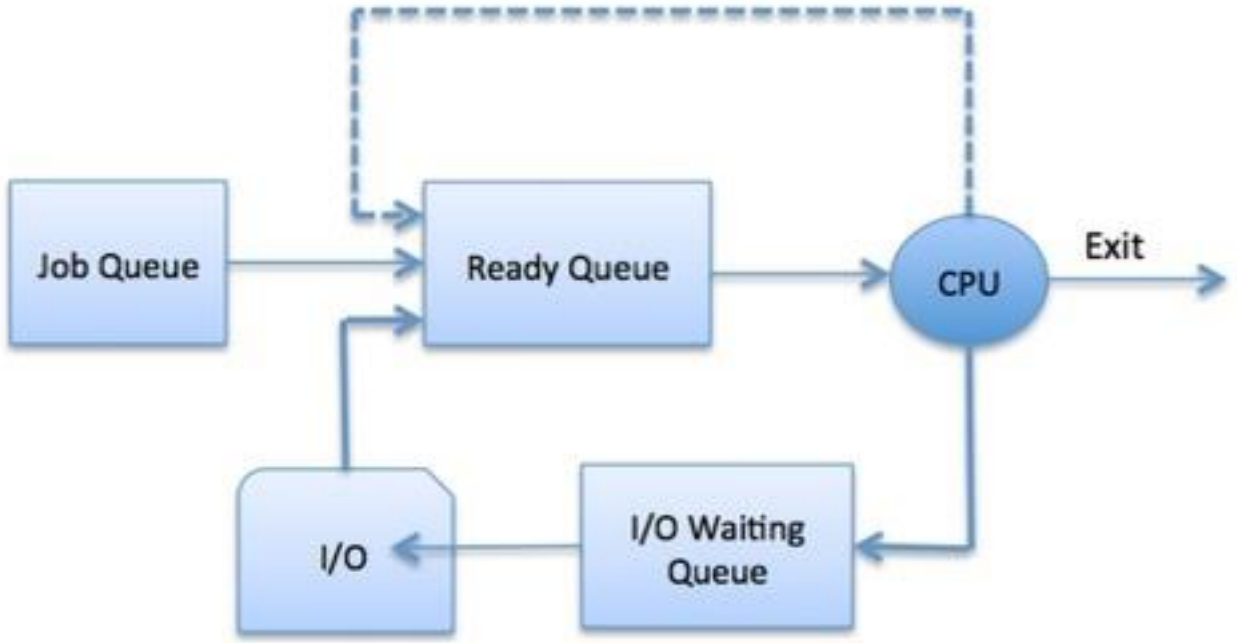
- Enqueue: Adds an item to the queue.
  - If the queue is full, then it is said to be an Overflow condition.
- Dequeue: Removes an item from the queue.
  - If the queue is empty, then it is said to be an Underflow condition.
- Front: Get the front item from the queue.
- Rear: Get the last item from the queue.

Applications of queue:

- Use in many applications where resources are shared among multiple users and served on a first come first serve basis.

Applications of queue:

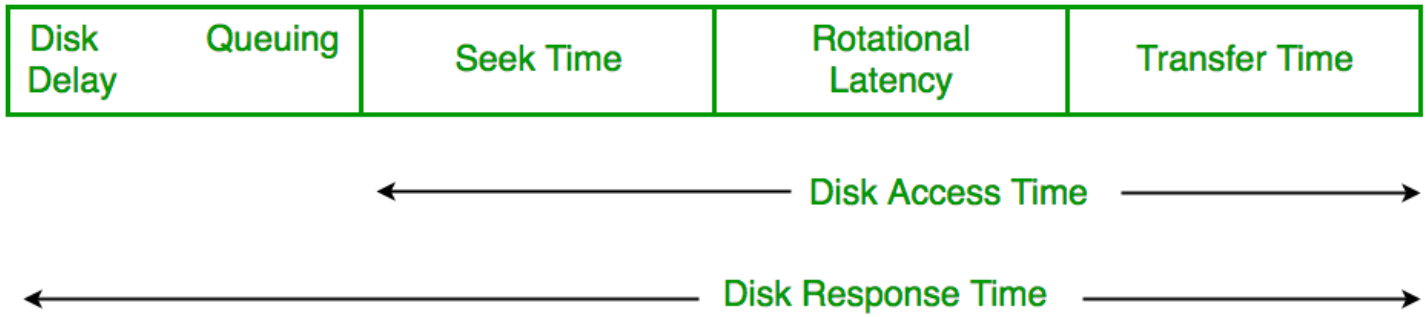
- CPU scheduling
  - Job queue – This queue keeps all the processes in the system.
  - Ready queue – This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
  - Device queues – The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

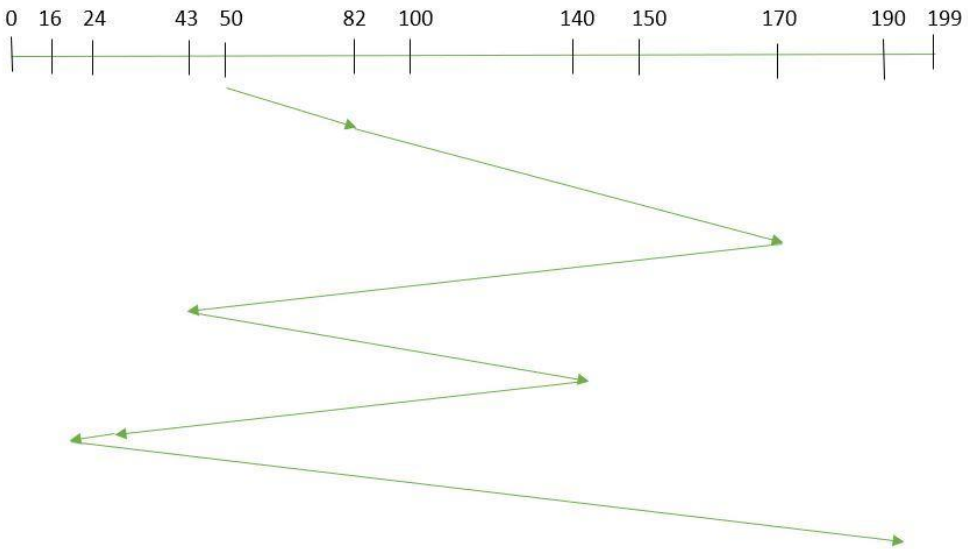
Applications of queue:

- Disk Scheduling.



Suppose the order of request is-
(82,170,43,140,24,16,190)

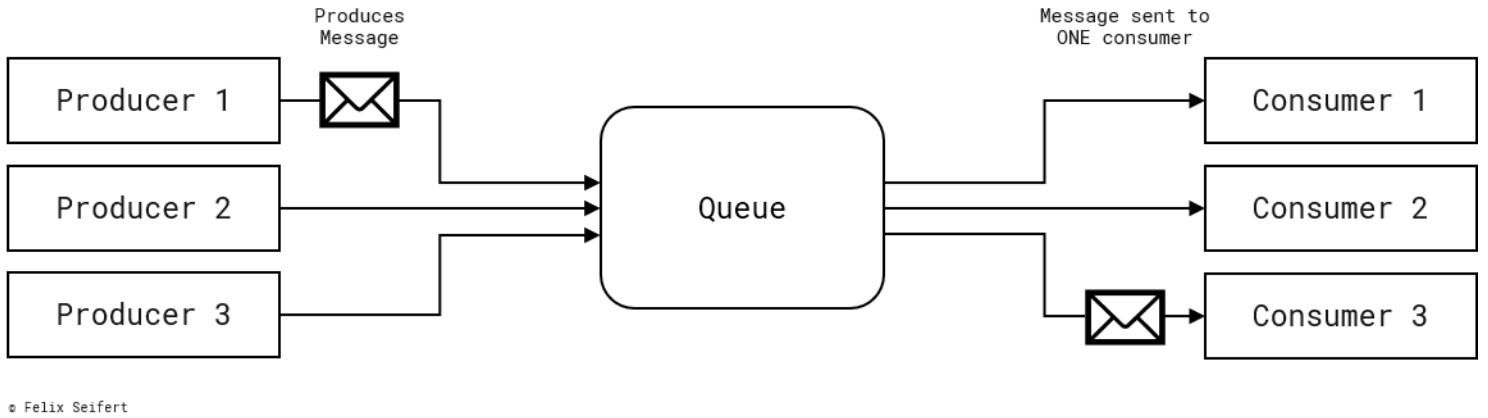
And current position of Read/Write head is :
50

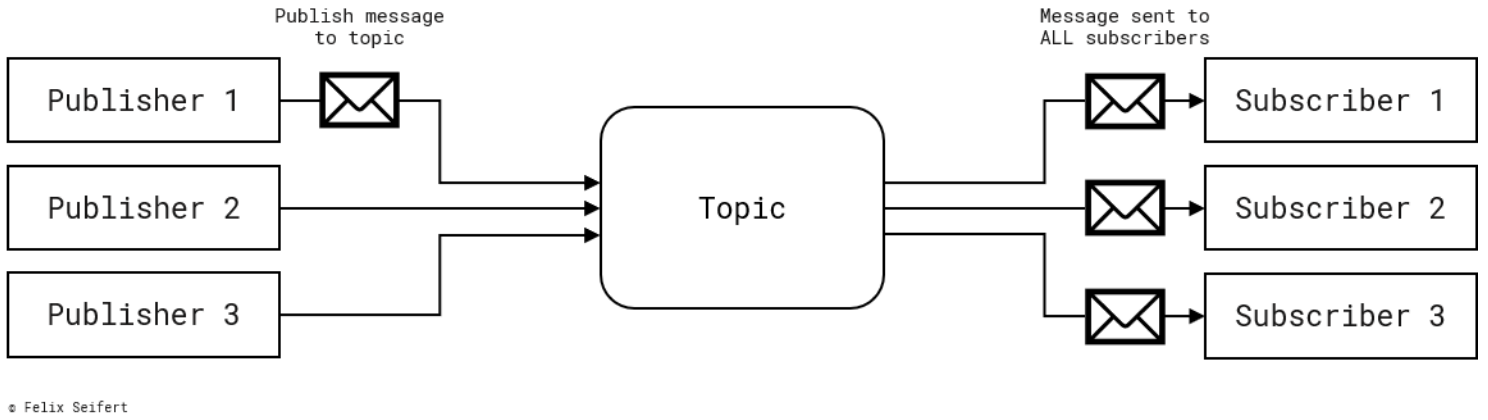


So, total seek time:
=(82-50)+(170-82)+(170-43)+(140-43)+(140-24)+(24-16)+(190-16)
=642

Applications of queue:

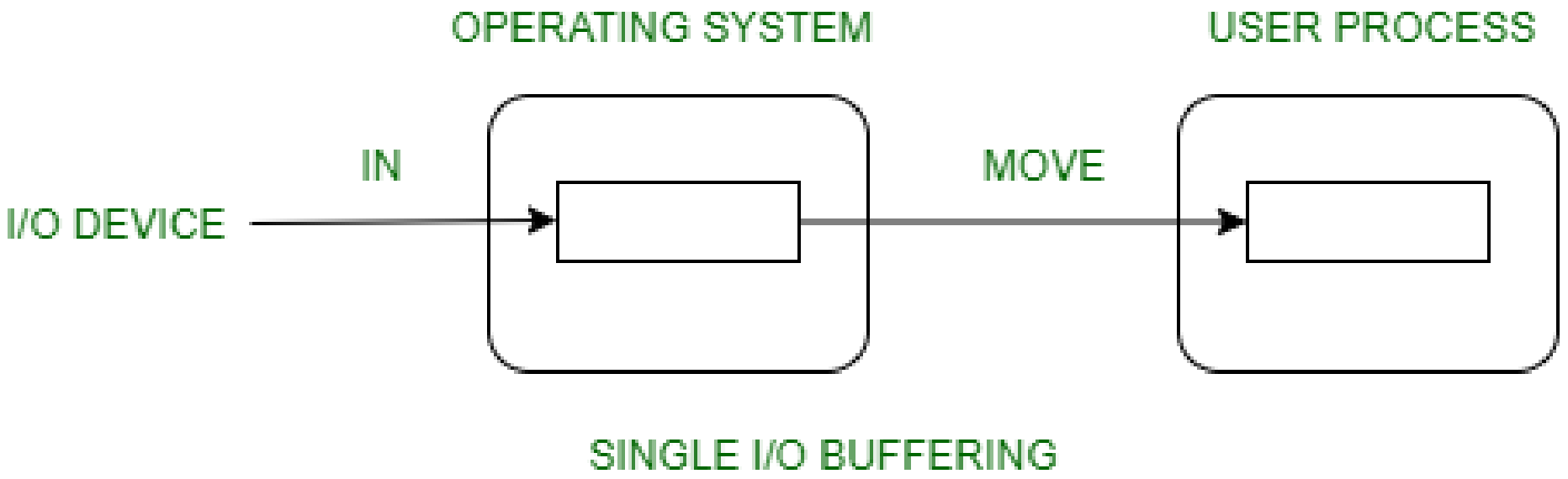
- Asynchronous data transfer (data not necessarily received at the same rate as sent) between two processes.
- streaming data





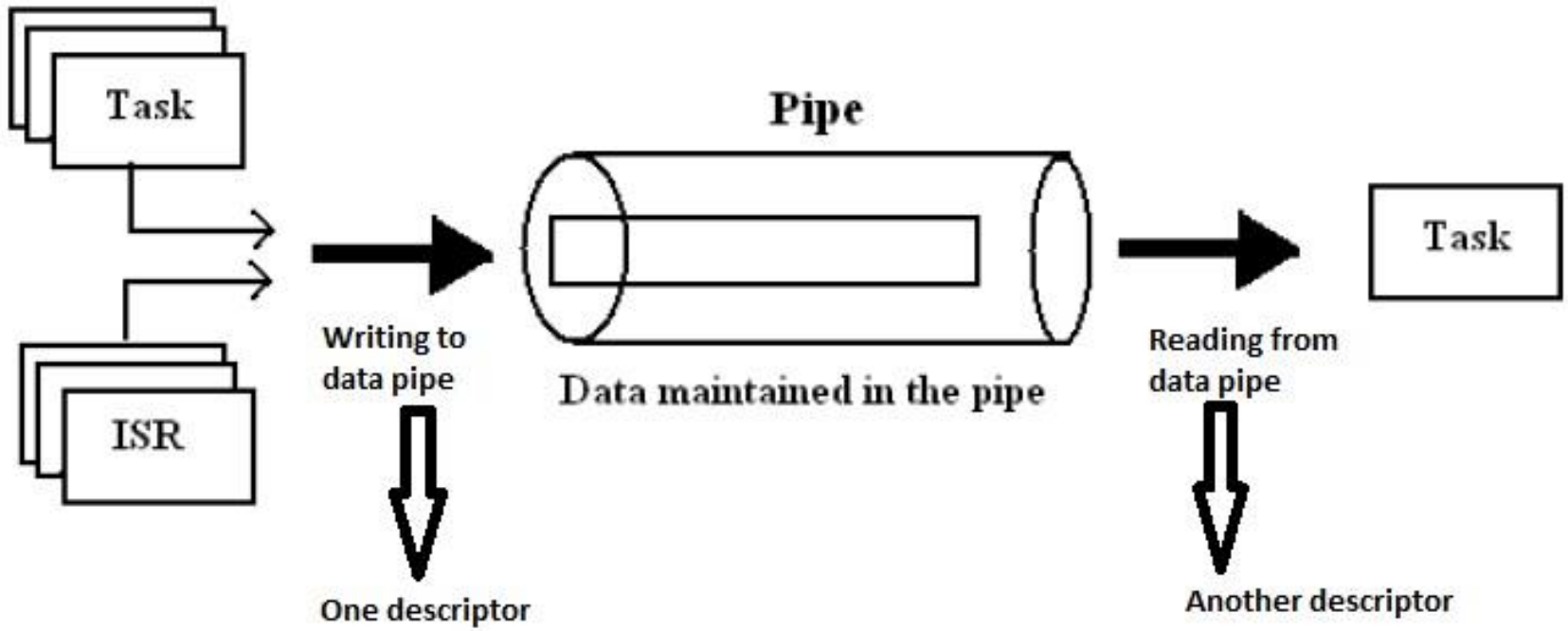
Applications of queue:

- IO Buffers



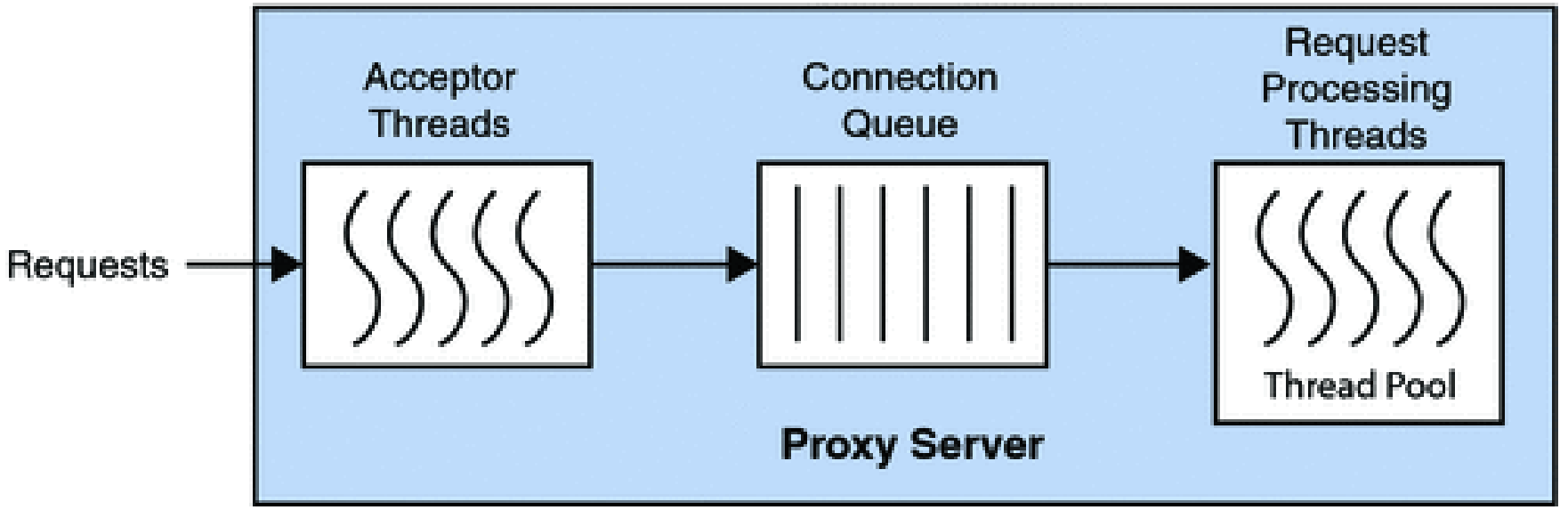
Applications of queue:

- Pipes



Applications of queue:

- Web traffic queue



Applications of queue:

- playlist in media players.
- Queue is used in operating systems for handling interrupts.
- It helps in serving requests on a single shared resource, like a printer, CPU task scheduling, etc.
- A real world example of queue
  - a single lane one way road, where the vehicle that enters first will exit first.
  - Cashier line in a store
  - People on an escalator

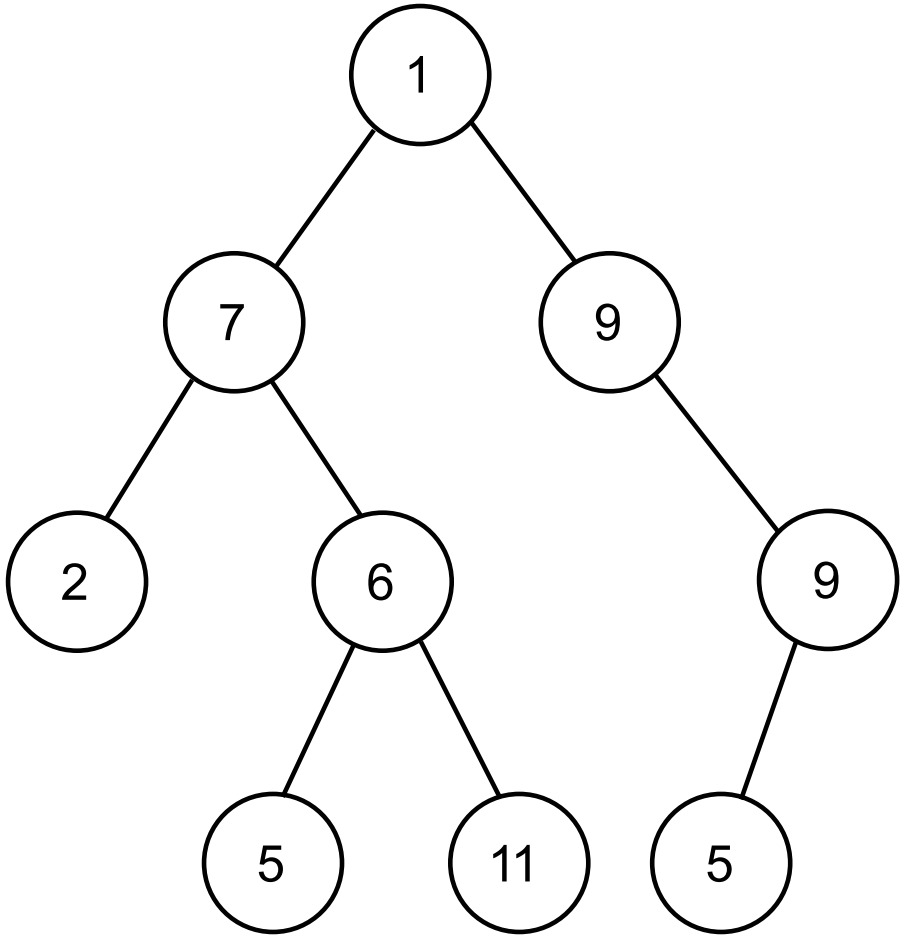
```
# Python program to
# demonstrate queue implementation
# using list

# Initializing a queue
queue = []

# Adding elements to the queue
queue.append('a')
queue.append('b')
queue.append('c')

print("Initial queue")
print(queue)

# Removing elements from the queue
print("\nElements dequeued from queue")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0))

print("\nQueue after removing elements")
print(queue)

# Uncommenting print(queue.pop(0))
# will raise and IndexError
# as the queue is now empty
```

---------- : Output : ----------

```
Initial queue
['a', 'b', 'c']

Elements dequeued from queue
a
b
c

Queue after removing elements
[]
```

Binary Tree

- trees are hierarchical data structures.
- Like link list
- A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.
- A Binary Tree is represented by a pointer to the topmost node in the tree.
- If the tree is empty, then the value of root is NULL.
- A Binary Tree node contains the following parts.
    1. Data
    2. Pointer to left child
    3. Pointer to the right child

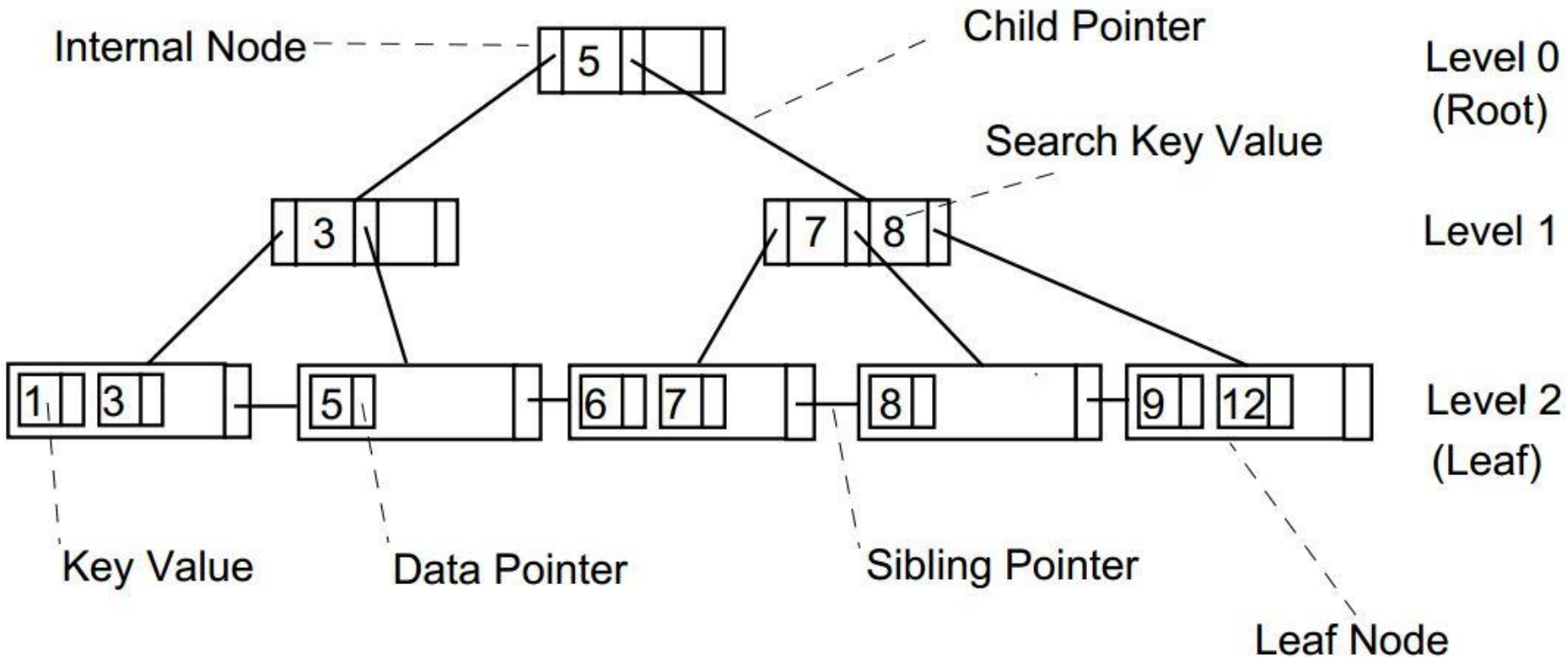


Applications of tree:

- B-Tree and B+ Tree are used to implement indexing in databases.
- B+ Tree Visualization : https://dichchankinh.com/~galles/visualization/BPlusTree.html

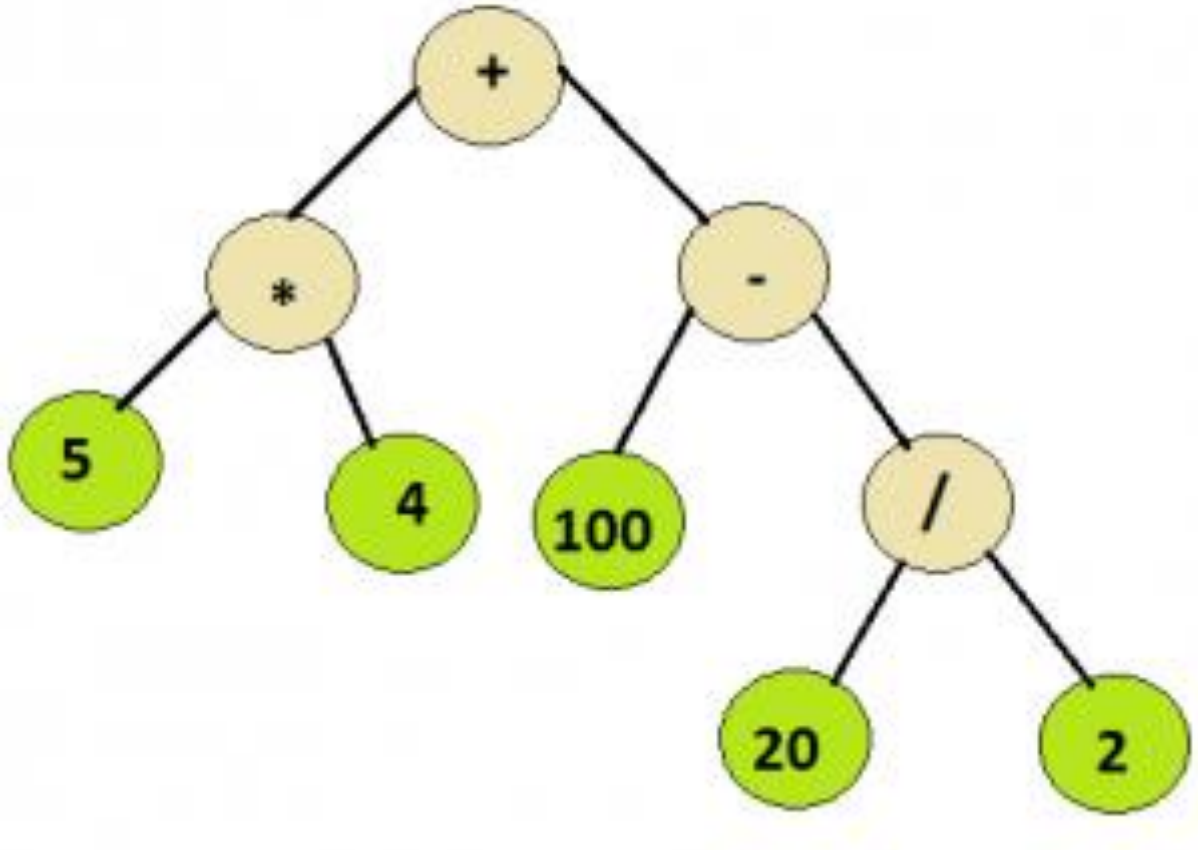
Applications of tree:

- B-Tree and B+ Tree are used to implement indexing in databases.
- B+ Tree Visualization : https://dichchankinh.com/~galles/visualization/BPlusTree.html



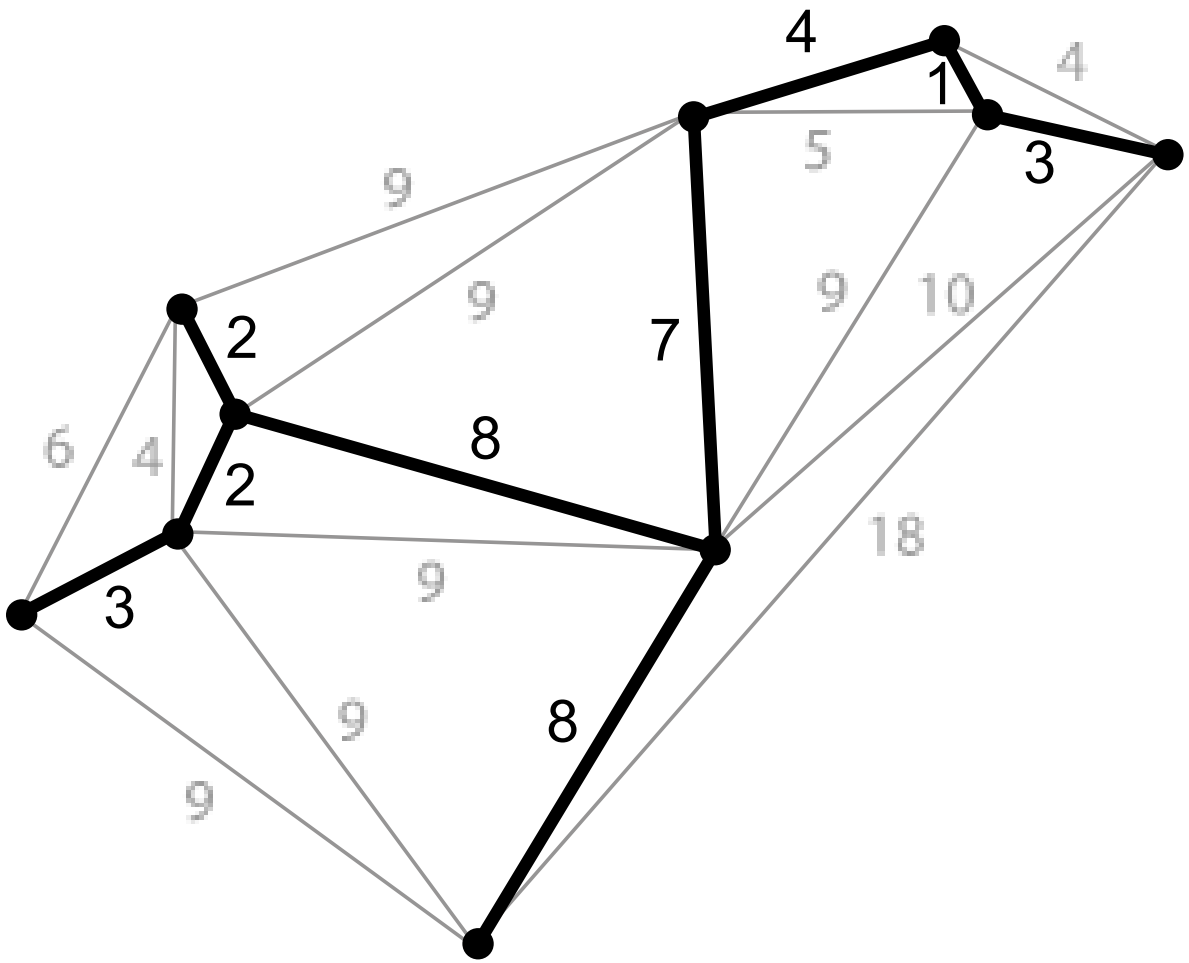
Applications of tree:

- Syntax Tree helps in scanning, parsing, generation of code and evaluation of arithmetic expressions in Compiler design.



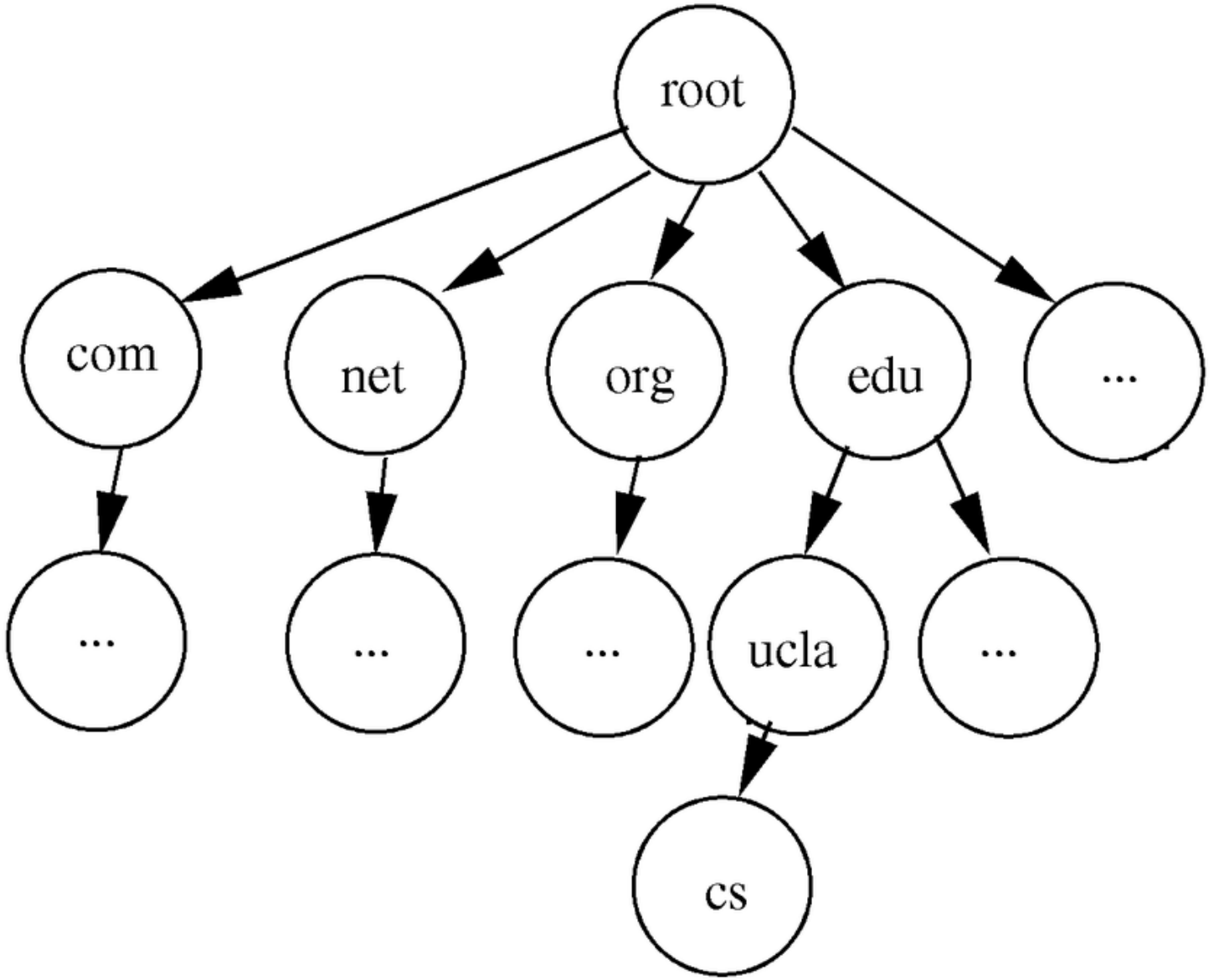
Applications of tree:

- Spanning trees are used in routers in computer networks.



Applications of tree:

- Domain Name Server also uses tree data structure.



Applications of tree:

- Real Life Applications of Tree:
- In real life, tree data structure helps in Game Development.
- It also helps in indexing in databases.
- Decision Tree is an efficient machine learning tool, commonly used in decision analysis. It has a flowchart-like structure that helps to understand data.
- The most common use case of a tree is any social networking site.

```
class Node:
  def __init__(self, data):
    self.left = None
    self.right = None
    self.data = data

  def insert(self, data):
# Compare the new value with the parent node
    if self.data:
      if data < self.data:
        if self.left is None:
          self.left = Node(data)
        else:
          self.left.insert(data) # recursive !!!
        elif data > self.data:
          if self.right is None:
            self.right = Node(data)
          else:
            self.right.insert(data) # recursive !!!
    else:
      self.data = data
```

```
# Print the tree
  def PrintTree(self):
    if self.left:
      self.left.PrintTree()
    print( self.data),
    if self.right:
      self.right.PrintTree()

# Use the insert method to add nodes
root = Node(12)
root.insert(6)
root.insert(14)
root.insert(3)
root.PrintTree()
```

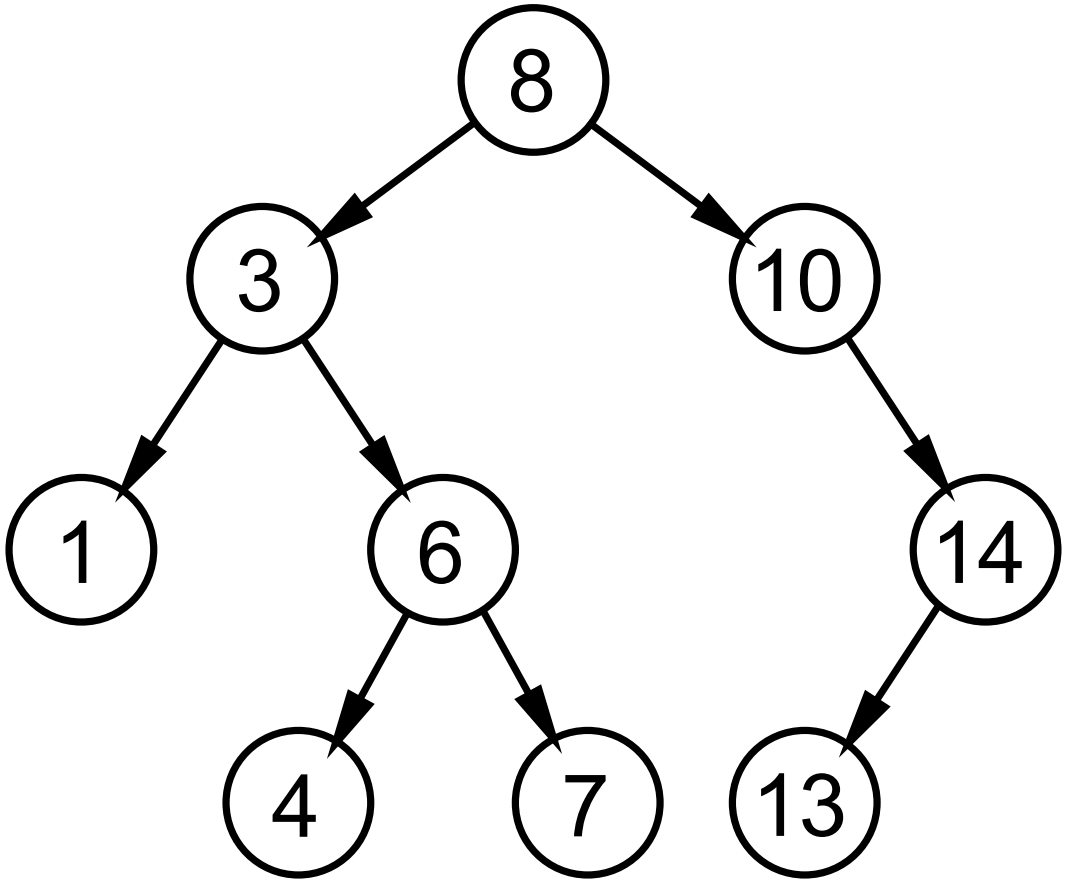
---------- : Output : ----------

3 6 12 14

Binary Search Tree

Binary Search Tree has following additional properties:

- The left subtree of a node contains only nodes with keys less than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree



Applications of tree:

- K-D Tree is a space partitioning tree used to organize points in K-dimensional space.
- BSTs are used for indexing and multi-level indexing.
- They are also helpful to implement various searching algorithms.
- It is helpful in maintaining a sorted stream of data.
- TreeMap and TreeSet data structures are internally implemented using self-balancing BSTs

```
class GFG :
        @staticmethod
        def main( args) :
                tree = BST()
                tree.insert(30)
                tree.insert(50)
                tree.insert(15)
                tree.insert(20)
                tree.insert(10)
                tree.insert(40)
                tree.insert(60)
                tree.inorder()
class Node :
        left = None
        val = 0
        right = None
        def __init__(self, val) :
                self.val = val
class BST :
        root = None
        def insert(self, key) :
                node = Node(key)
                if (self.root == None) :
                        self.root = node
                        return
                prev = None
                temp = self.root
                while (temp != None) :
                        if (temp.val > key) :
                                prev = temp
                                temp = temp.left
                        elif(temp.val < key) :
                                prev = temp
                                temp = temp.right
                if (prev.val > key) :
                        prev.left = node
                else :
                        prev.right = node
        def inorder(self) :
                temp = self.root
                stack = []
                while (temp != None or not (len(stack) == 0)) :
                        if (temp != None) :
                                stack.append(temp)
                                temp = temp.left
                        else :
                                temp = stack.pop()
                                print(str(temp.val) + " ",
end ="")
                                temp = temp.right

if __name__=="__main__":
        GFG.main([])
```

---------- : Output : ----------

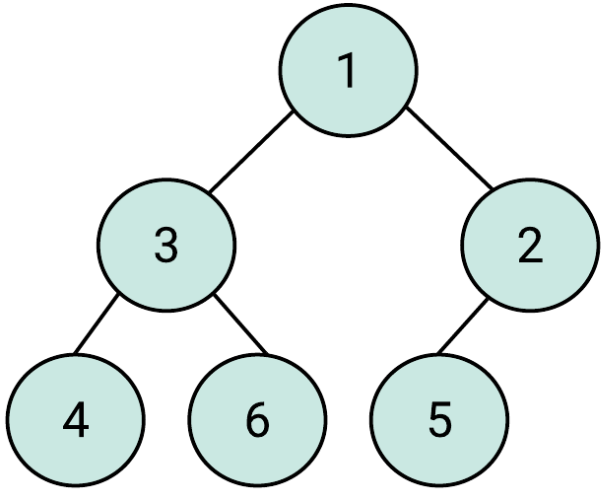
10 15 20 30 40 50 60

Heap

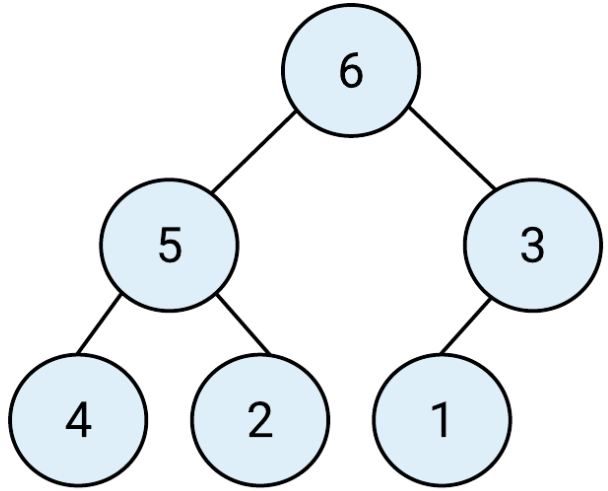
A Heap is a special Tree-based data structure in which the tree is a complete binary tree.

Generally, Heaps can be of two types:

- Max-Heap: Root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.
- Min-Heap: Root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.



Min heap



Max Heap

Applications of heap:

- Priority Queues can be implemented using heaps. The root of a heap always contains the maximum or the minimum value, based on the heap type. Therefore, a min-priority queue is implemented using a minheap. A max-priority queue is implemented using a maxheap. The element with the highest priority can be retrieved in O(1) time.

- Statistics – If we want to get ordered statistics, heaps serve as a great choice. If we want the kth smallest or largest element, we can pop the heap k times to retrieve them.

- Heaps are used in implementing various graph algorithms like Dijkstra’s algorithm and Prim’s algorithm.

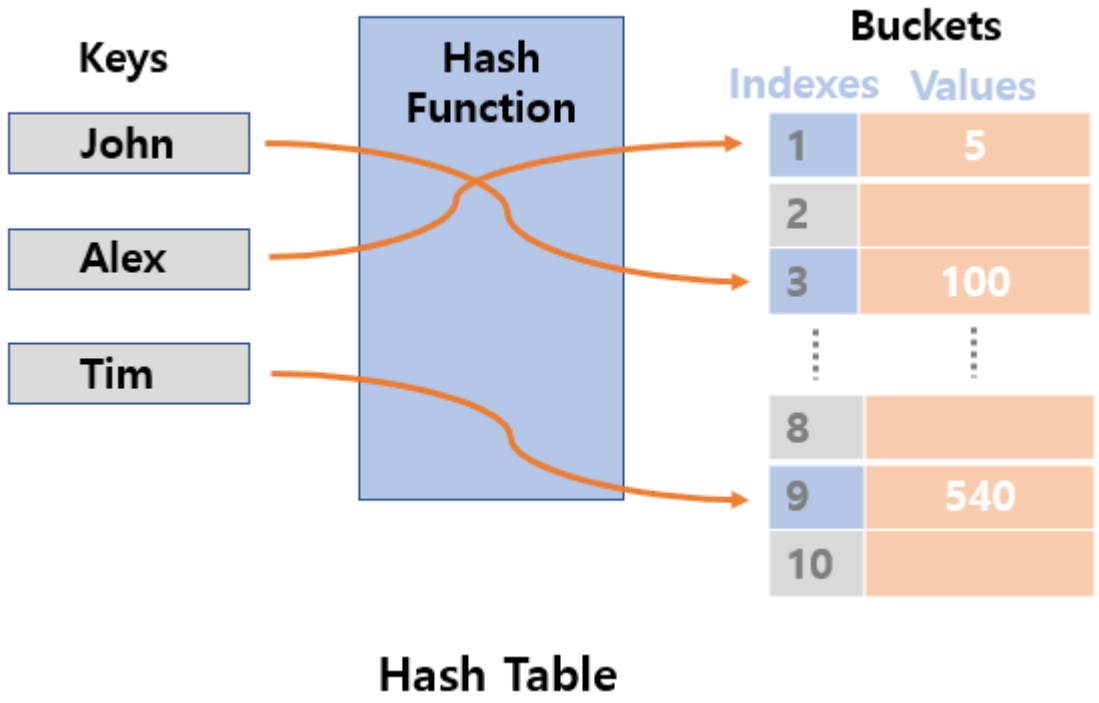
```
# Python code to demonstrate working of
# heapify(), heappush() and heappop()

# importing "heapq" to implement heap queue
import heapq

# initializing list
li = [5, 7, 9, 1, 3]

# using heapify to convert list into heap
heapq.heapify(li)

# printing created heap
print ("The created heap is : ",end="")
print (list(li))

# using heappush() to push elements into heap
# pushes 4
heapq.heappush(li,4)

# printing modified heap
print ("The modified heap after push is : ",end="")
print (list(li))

# using heappop() to pop smallest element
print ("The popped and smallest element is :
",end="")
print (heapq.heappop(li))
```

---------- : Output : ----------

```
The created heap is : [1, 3, 9, 7, 5]
The modified heap after push is : [1, 3, 4, 7, 5, 9]
The popped and smallest element is : 1
```

Hashing

- Hashing is designed to use a Hash function to map a given value with a particular key for faster access of elements.
- Let a hash function H(x) maps the value x at the index x%10 in an Array. For example, if the list of values is [11, 12, 13, 14, 15] it will be stored at positions {1, 2, 3, 4, 5} in the array or Hash table respectively.
- Python use hash function in dictionary data type



Hash function

```
# Python 3 code to demonstrate
# working of hash()

# initializing objects
int_val = 4
str_val = 'GeeksforGeeks'
flt_val = 24.56

# Printing the hash values.
# Notice Integer value doesn't change
# You'l have answer later in article.
print("The integer hash value is : " +
str(hash(int_val)))
print("The string hash value is : " + str(hash(str_val)))
print("The float hash value is : " + str(hash(flt_val)))
```

```
---------- : Output : ----------

The integer hash value is : 4
The string hash value is : -5570917502994512005
The float hash value is : 1291272085159665688
```

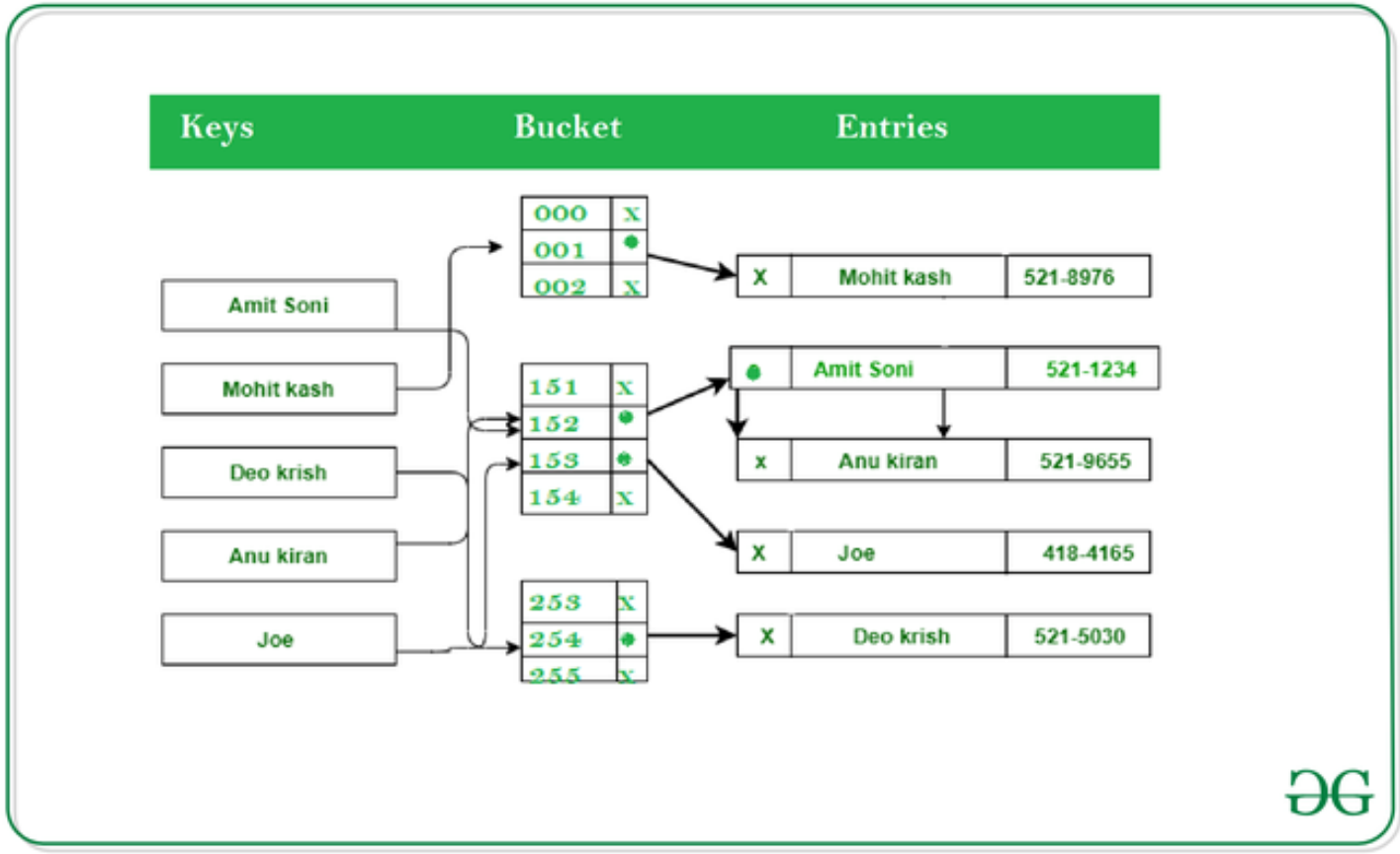
Python use hash in dictionary data type

```
# code
import sys

d = {}
print(sys.getsizeof(d))
```

---------- : Output : ----------

240

- Empty dict consume 240 bytes



Python use hash in dictionary data type

```
import sys

d = {}
d['a'] = 'a' * 100000
print("Size of dictionary ->", sys.getsizeof(d))
print("Size of a ->", sys.getsizeof('a'))

--------------- : Output : ---------------
Size of dictionary -> 240
Size of a -> 50
```

Python use hash in dictionary data type

```
import sys

d = {}
d['python'] = 1
for key in list(d.keys()):
  d.pop(key)

print(len(d))
print(sys.getsizeof(d))
```

```
---------- : Output : ----------
0
240
```

Python use hash in dictionary data type

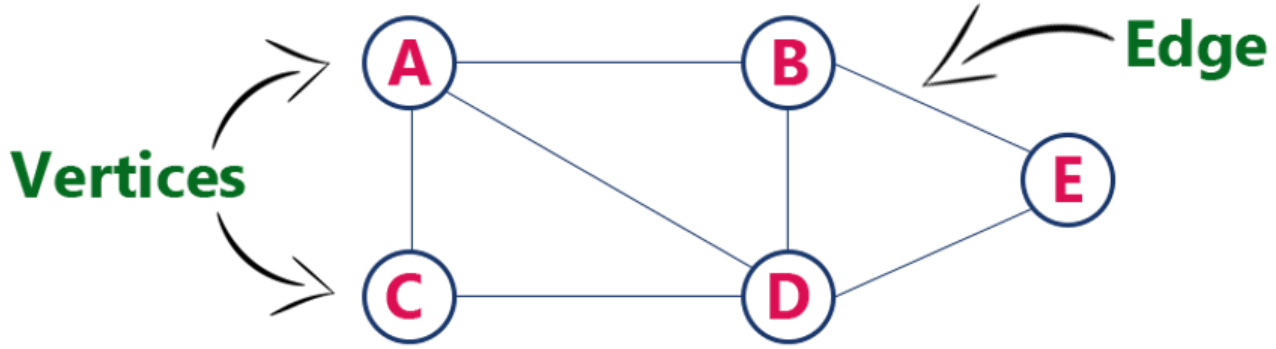
```
import sys

d = {}
d['python'] = 1
for key in list(d.keys()):
  d.pop(key)

print(len(d))
d.clear()
print(sys.getsizeof(d))
```

```
---------- : Output : ----------
0
72
```

Applications of hashing:

- Fast search data storage
- game maps
- jump table to emulate a switch type statement
- build indexes of content
- text analysis function

Graph



- A graph is comprised of objects connected by lines.
- we refer to those objects and lines as vertices and edges.
- Represent data and their relationship to each other
- Two common properties of edges are weights and direction.

Applications of Graph

- The operating system uses Resource Allocation Graph.
- In World Wide Web where the web pages represent the nodes.
- In Google Maps , cities are located as vertices and paths connecting those vertices are located as edges of the graph.
- In social network , every person on the network is a node, and all of their friendships on the network are the edges of the graph.
- A graph is also used to study molecules in physics and chemistry.

Create graph with dictionary

```
# Create the dictionary with graph elements
graph = {
   "a" : ["b","c"],
   "b" : ["a", "d"],
   "c" : ["a", "d"],
   "d" : ["e"],
   "e" : ["d"]
}
# Print the graph
print(graph)
```

---------- : Output : ----------

```
{'c': ['a', 'd'], 'a': ['b', 'c'], 'e': ['d'], 'd': ['e'], 'b': ['a', 'd']}
```

Create graph with dictionary

```
class graph:
   def __init__(self,gdict=None):
      if gdict is None:
         gdict = []
      self.gdict = gdict
# Get the keys of the dictionary
   def getVertices(self):
      return list(self.gdict.keys())
# Create the dictionary with graph elements
graph_elements = {
   "a" : ["b","c"],
   "b" : ["a", "d"],
   "c" : ["a", "d"],
   "d" : ["e"],
   "e" : ["d"]
}
g = graph(graph_elements)
print(g.getVertices())
```

---------- : Output : ----------

```
['d', 'b', 'e', 'c', 'a']
```

Create graph with dictionary

```
class graph:
  def __init__(self,gdict=None):
    if gdict is None:
      gdict = {}
    self.gdict = gdict

  def edges(self):
    return self.findedges()
# Find the distinct list of edges
  def findedges(self):
    edgename = []
    for vrtx in self.gdict:
      for nxtvrtx in self.gdict[vrtx]:
        if {nxtvrtx, vrtx} not in edgename:
          edgename.append({vrtx, nxtvrtx})
    return edgename
# Create the dictionary with graph elements
graph_elements = {
  "a" : ["b","c"],
  "b" : ["a", "d"],
  "c" : ["a", "d"],
  "d" : ["e"],
  "e" : ["d"]
}
g = graph(graph_elements)
print(g.edges())

---------- : Output : ----------
[{'b', 'a'}, {'b', 'd'}, {'e', 'd'}, {'a', 'c'}, {'c', 'd'}]
```

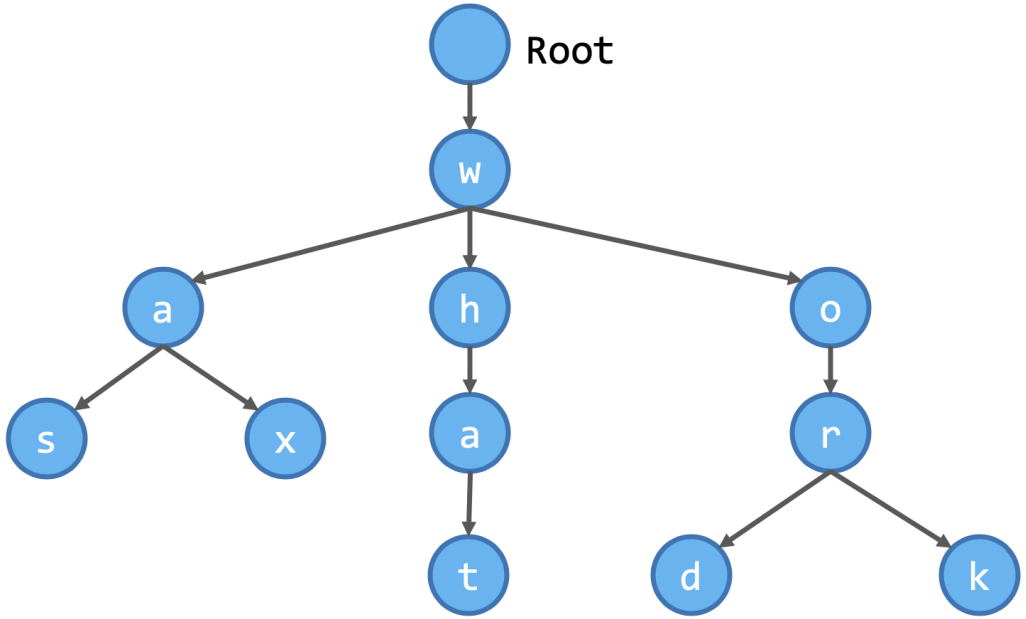
Create graph with dictionary

```
# Add the vertex as a key
  def addVertex(self, vrtx):
    if vrtx not in self.gdict:
      self.gdict[vrtx] = []


# Add the new edge
  def AddEdge(self, edge):
    edge = set(edge)
    (vrtx1, vrtx2) = tuple(edge)
    if vrtx1 in self.gdict:
      self.gdict[vrtx1].append(vrtx2)
    else:
      self.gdict[vrtx1] = [vrtx2]
# List the edge names
  def findedges(self):
    edgename = []
```

```
for vrtx in self.gdict:
  for nxtvrtx in self.gdict[vrtx]:
    if {nxtvrtx, vrtx} not in edgename:
      edgename.append({vrtx, nxtvrtx})
 return edgename
```

Trie

The word “trie” actually is derived from the word “reTRIEval”. Retrieval meaning “to get something back from somewhere”.



Trie is a tree-like data structure made up of nodes. Nodes can be used to store data. Each node may have none, one or more children. When used to store a vocabulary, each node is used to store a character, and consequently each "branch" of the trie represents a unique word. The following figure shows a trie with five words (was, wax, what, word, work) stored in it.

```
class Trie(object):
  def __init__(self):
    self.child = {}
  def insert(self, word):
    current = self.child
    for l in word:
      if l not in current:
        current[l] = {}
      current = current[l]
    current['#']=1
  def search(self, word):
    current = self.child
    for l in word:
      if l not in current:
        return False
      current = current[l]
    return '#' in current
  def startsWith(self, prefix):
    current = self.child
    for l in prefix:
      if l not in current:
        return False
      current = current[l]
    return True
ob1 = Trie()
ob1.insert("apple")
print(ob1.search("apple"))
print(ob1.search("app"))
print(ob1.startsWith("app"))
ob1.insert("app")
print(ob1.search("app"))
```

---------- : Output : ----------

```
True
False
True
True
```

Application of trie

- Auto-correct has been one of the most popular uses for this data structure. Your device or the app you use (for example Google docs) uses tries for its predictive typing or auto-correct feature.

- Spell check is another common application of tries. Tries can be used to store words and then, search for specific words through the data structure.

- Suggestions for the words you misspelled (the ones that are close) will come up for you to select when the spell-check feature is enabled.