Let’s Go with Algo

By Melakesleam Moges

09-12-2023

BLAST FROM THE PAST

Quick Quiz

1. **What are the three Limited Resources that affect Algorithms?**
2. **List three common tasks performed in computing. Mention if each task is primitive or derived.**
3. **Which of the search algorithms we studied follows this steps:**
   a. Divide the list into √n groups
   b. Go to the first group
   c. Check the boundary values to see if the target could exist in the group
   d. If it can exist in the group, check one by one
   e. If not, go to the next group and repeat step c.

BLAST FROM THE PAST

Quick Quiz Answers

1. What are the three Limited Resources that affect Algorithms?
   a. **Processing Power (Time), Memory (Space), Network (Speed and Bandwidth)**
2. List three common tasks performed in computing. Mention if each task is primitive or derived.
   a. **Search(primitive), Sort(primitive), Insert(derived), Update(derived), Delete(derived)**
3. Which of the search algo**rithms we studied follows this steps**
   a. **Jump Search**

LECTURE 5

Fundamental Data Structures

NEED FOR DATA STRUCTURES

We need Data Structures to efficiently

- Store data
- Organize data to speed up access and update process
- Improve the performance of Algorithms

TIME VERSUS SPACE

To improve the time efficiency of an algorithm, it is often at the cost of using more memory space.

As technology progresses, storage is becoming more abundant compared to the speed of processing

Common Data Structures

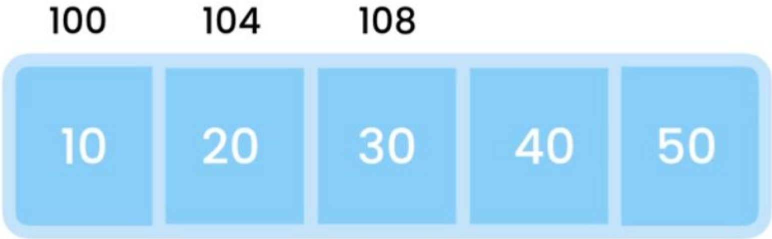
- Array
- Linked List
- Queue
- Stack

Vital Data Structures to Research (*intermediate level*)

- Hash Table(Map)
- Tree (Binary, AVL, Red-Black)
- Heaps
- Tries
- Graphs

Array

Arrays are the simplest DS. They are dimensional consecutive storage spaces that depict physical storage space in memory and storage spaces.

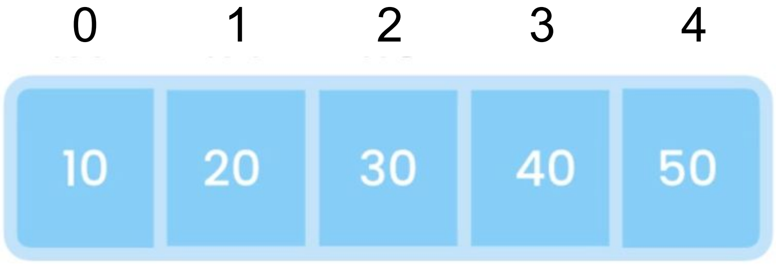


- ***Physical memory*** *stores one data type - fixed size - eg.* ***int*** *needs 4B*
- *Stores in sequence - no gap*
- *Has memory address*
  Root Address + (position * size)

- ***Array*** *holds items of one type*
- *Size depends on number of items*
- *Each item storage has index starting from 0. Refers to address in Physical memory*

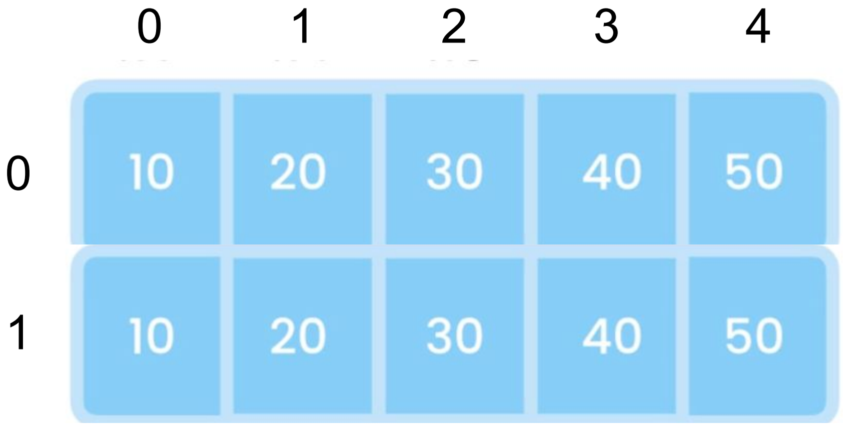
Array - usage

- Store list of items to easily access a value by **calculating its location** or **sequentially accessing the index** of each item
- Can store one dimensional list or multi-dimensional list
- Used to implement other Data Structures



0	1	2	3	4
10	20	30	40	50

One Dimension



	0	1	2	3	4
0	10	20	30	40	50
1	10	20	30	40	50

Two Dimension [row,col]

Array - characteristics

Advantages

- Look up of an item at an index is very fast
- Ideal for storing of items of known list size
- Appending new items is very fast

Disadvantages

- If the size of the list is unknown or fluctuates,
  - Size too small leads to memory wastage
  - Size too large leads to waste of time creating new array and copying elements to it
- Adding or Removing items at the beginning or the middle requires shifting of items to the right which adds costly operations each time

Array - implementation

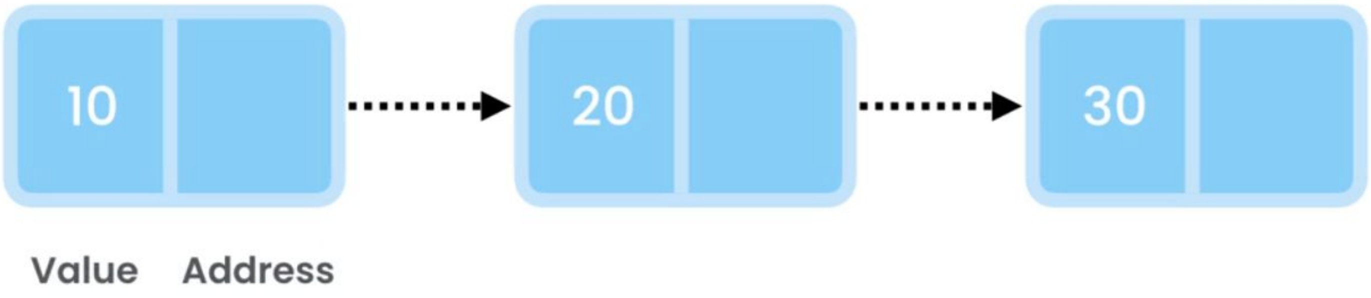
Java	**JavaScript***	**Python***
Declaration and Instantiation		
int[] nums = new int[10];	nums = [1, 5, 10];	nums = [1, 5, 10]
int[] nums = {1, 5, 10};	variable = [1, '1', "hello"];	variable = [1, '1', "hello"]
Add / Update Item in the Middle		
nums[2] = 7;	nums[2] = 7;	nums.insert(2, 7)

*** this language doesn't provide a pure implementation of an Array**

Linked List

Linked Lists are composed of ***nodes*** *that store the* ***value of the item*** *and the* ***address of the node*** *that has the next item on the list.*

Each node points to (references) the node that holds the next item on the list

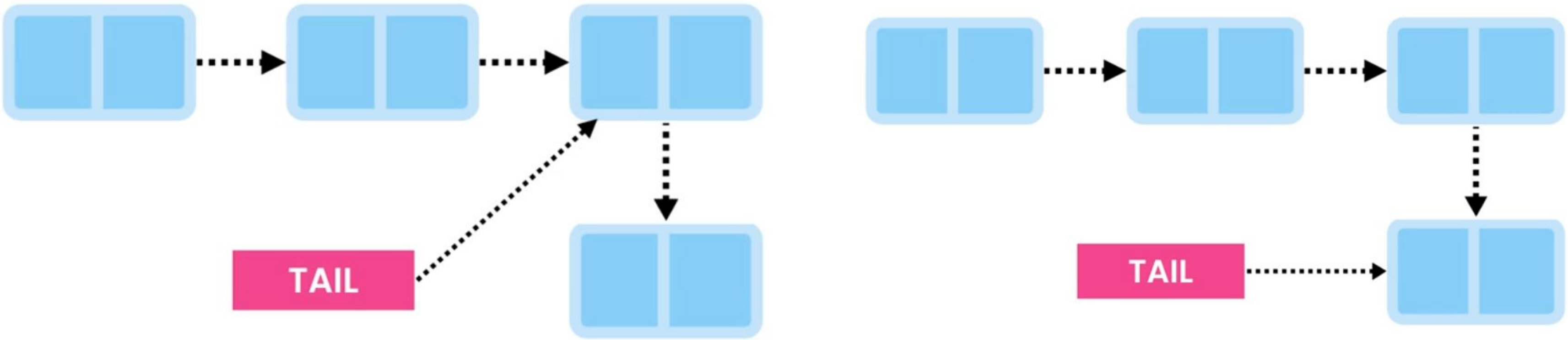


Head

Tail

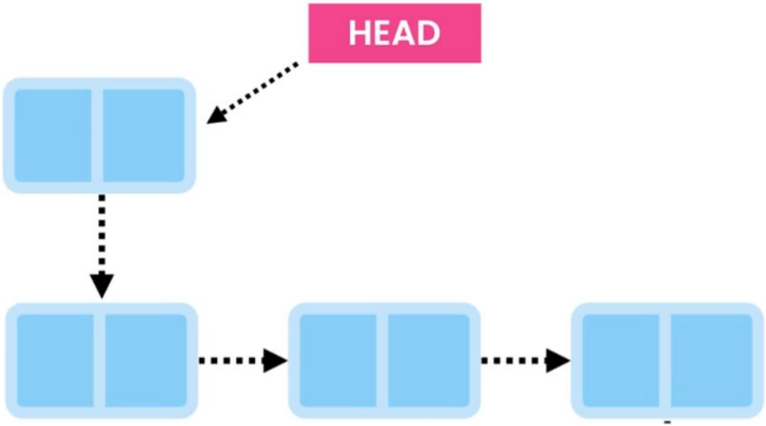
Linked List

Insert At the End

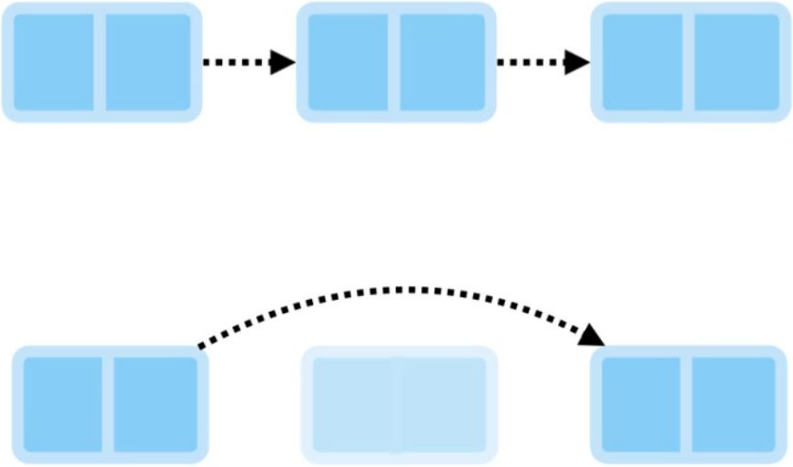


Linked List

Insert At the Beginning



Delete from the Middle



Linked List

Creating a Linked list

1. Declare a Node class
2. Declare a LinkedList class with ‘head’ field
3. Create insert methods
4. Create update methods
5. Create delete methods

Linked List

LinkedList.py

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

class LinkedList:

  def __init__(self):
    self.head = None
```

Linked List

```
def insertAtBegin(self, data):
    # create new node with data
    new_node = Node(data)
    # if list is empty
    if self.head is None:
        self.head = new_node
        return
    # if list is not empty
    else:
        new_node.next = self.head
        self.head = new_node
```

Linked List

```python
def insertAtIndex(self, data, index):
    new_node = Node(data)
    current_node = self.head
    position = 0
    if position == index:
        self.insertAtBegin(data)
    else:
        # traversing the nodes
        while(current_node != None and position+1 != index):
            position = position+1
            current_node = current_node.next

        if current_node != None:

            new_node.next = current_node.next
            current_node.next = new_node
        else:
            print("Index not present")
```

Linked List

```
def remove_first_node(self):
    if(self.head == None):
      return
    self.head = self.head.next
```

Linked List

Using the linked list

```
ll = LinkedList()
ll.insertAtBegin(5)
ll.remove_first_node()
```

Linked List

Exercise

- `inserAtEnd`
- `updateNode`
- `remove_last_node`
- `remove_at_index`
- `remove_data`
- `clear`

Linked List - usage

- Store list of items to **easily add and remove** an item at **any location**
- Can store one dimensional list or multi-dimensional list
- Used to implement other Data Structures

Linked List - characteristics

Advantages

- Add and remove an item once located is fast
- Can store dynamically sized list that grows and shrinks frequently

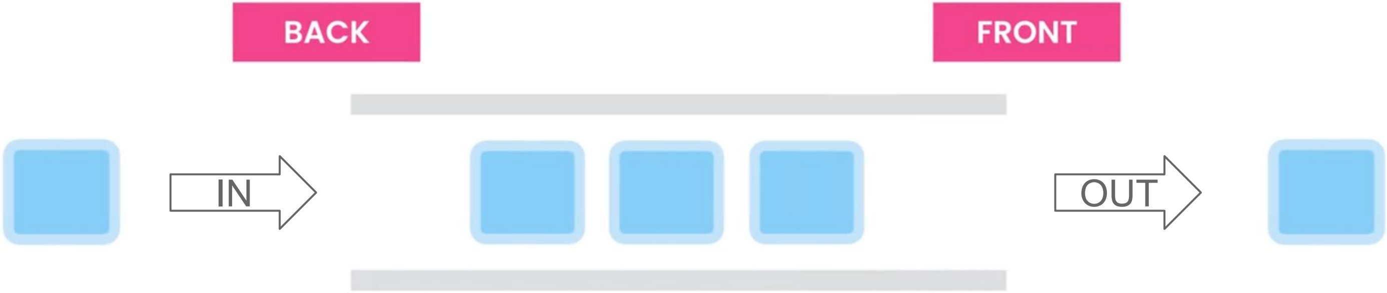
Disadvantages

- Look up of an item by value or by index is slow as it requires traversing the nodes until the value is found or the index is reached starting from the ***head***.

Queue

This is a special type of list that exhibits certain behavior and provides specific functionality referred to as First-In First Out (FIFO)

- Items are **Always** added **at the end.**
- Items are **Always** removed **from the front.**



Queue - usage

- As the name suggests, use this DS to implement First come First served functionality.
- This is used to manage the sharing of limited resource to significantly more users.
- Examples of Systems:
  - Printers,
  - Web servers,
  - Operating systems,
  - Live support systems,
  - Restaurants

Queue - implementation and operations

We can use Arrays(*Circular*) or LinkedLists(*Double Ended*) to implement Queues

- Enqueue
  - Add at the end
- Dequeue
  - Remove from the Beginning and return value.
- Peek
  - Return the value at the Beginning without removing it.
- isEmpty
  - Returns true if the value at the Head pointer is null or empty
- isFull
  - Returns true if the queue size is limited like when using an Array or specifically defined to be of a certain size

Queue - implementation and operations

Queue.py

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

class Queue:
  def __init__(self) -> None:
     self.head = None
     self.tail = None
```

Queue - implementation and operations

Queue.py

```
def enqueue(self, data):
    new_node = Node(data)
    if self.tail == None:
      self.head = self.tail = new_node
    else:
      self.tail.next = new_node
      self.tail = new_node
```

Queue - implementation and operations

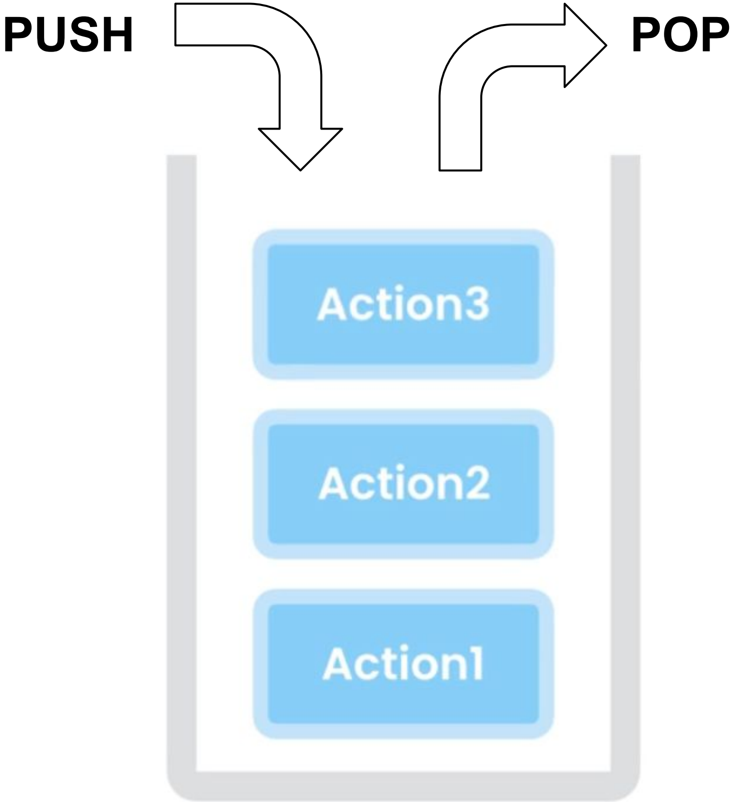
Queue.py

```
  def dequeue(self):
    if self.head == None:
      raise ValueError('dequeue called on an empty
queue')
    removed_node = self.head
    if self.head.next != None:
      self.head = self.head.next
    else:
      self.head = self.tail = None
   Removed_node.next = None #clean up
    return removed_node.data
```

Stack

This is another special type of list that exhibits certain behavior and provides specific functionality referred to as Last-In First Out (LIFO)

- Items are **Always** added **and** removed **from the TOP.**
- To reach the bottom of the stack, we have to remove all the elements in the stack



Stack - usage

- Has many functionalities in programming.
- Examples:
  - Implement the undo feature,
  - Build compilers(eg syntax checking),
  - Evaluate expressions (eg 1 + 2 * 3),
  - Build navigation (eg forward/back),
  - To trace the method calls in a programming language

Stack - implementation and operations

We can use Arrays or LinkedLists(*Single Ended*) to implement Stacks

- push(item)
  - Add at the Top
- pop()
  - Remove from the Top and return value.
- peek()
  - Return the value at the Top without removing it.
- isEmpty
  - Returns true if the stack is empty

Stack - implementation and operations

Stack.py

```
class Stack:
  def __init__(self, size) -> None:
    self.storage = []
    self.size = size
    self.top = 0

  def push(self,data):
    if self.top >= self.size:
      raise ValueError('The stack is full.')
    self.top += 1
    self.storage.append(data)
```

Stack - implementation and operations

Stack.py

```
def pop(self):
   if self.isEmpty():
     raise ValueError('The stack is empty.')
   self.top -= 1
   return self.storage.pop()

 def isEmpty(self):
   return self.top <= 0
```

Stack - implementation and operations

Stack.py

```
stack = Stack(3)
stack.push(5)
stack.push(6)
stack.push(7)
# stack.push(8)
print(stack.pop())
print(stack.pop())
print(stack.pop())
print(stack.pop())
```

Team Challenge

I have a list of names in a certain order stored in an Array:

[‘Melat’, ‘John’, ‘Yeabnat’, ‘Sisay’, ‘Tikdem’]

I wish to reverse the order of this list.

Task: using only the Data Structures we learnt today, write an **Algorithm** and its **Implementation** (in any Language you like) that will reverse this list of names.

Submit your solution with you names on your or your partner’s github account.
Due date: Dec. 15