




- 04 - Programming Foundations: Algorithms

 Certificate	CertificateOfCompletion_Programming_Foundations_Algorithms.pdf
 Completed Sections	8
 Course Links	https://www.linkedin.com/learning/programming-foundations-algorithms?resume=false
 Goal Sections	8
 Quick note	
 Start and Finish Date	
 Thumbnail	
 Time	1h 45m 30s
 Understanding %	
 نسبة الإنتهاء	 Done

Overview

▼ What are **algorithms**

- An algorithm is used to solve a specific problem by following a sequential set of steps.
- Algorithm Characteristics
 - Algorithm complexity
 - Space complexity: How much memory does it require?
 - Time complexity: How much time does it take to complete?
 - Inputs and output

- What does the algorithm accept, and what are the results?
- Classification
 - Serial / Parallel or exact

▼ Common algorithms in programming

- Search algorithms
 - Find specific data in a structure
 - For example , a substring within a string
- Sorting algorithms
- Computational algorithms
- Collection algorithms
 - Work with collections of data (count specific items , navigate among data elements , filter out unwanted dat, etc)
 - Example

```
def gcd(a, b):
    while( b != 0):
        t = a
        a = b
        b = t % b

    return a

# try out the function with a few examples
print(gcd(60, 96)) # should be 12
print(gcd(20, 8))  # should be 4
```

▼ Measuring algorithm performance

- Measure how an algorithm responds to dataset
- Big-O notation
 - Classifies performance as the input size grows
 - “O” indicates the order of operation: time scale to perform an operation
 - Many algorithms and data structures have more than one O
 - Inserting data, searching for data , deleting data

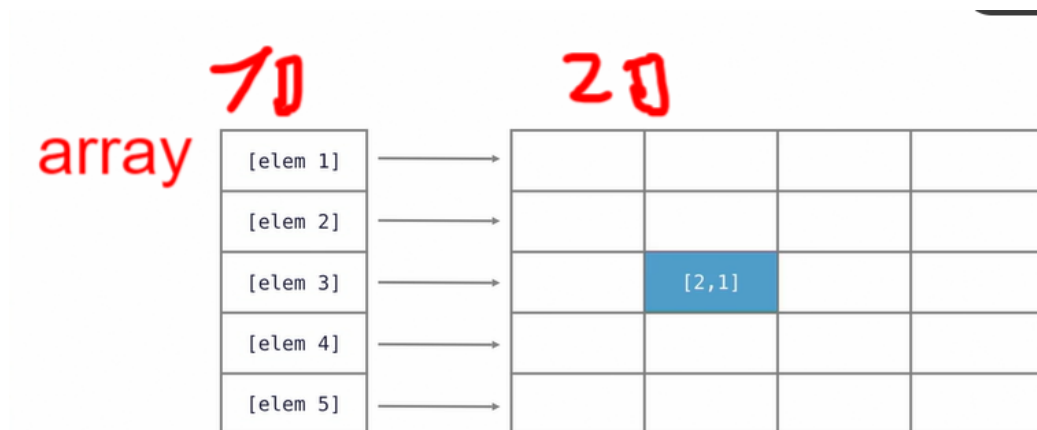
Notation	Description	Example
$O(1)$	Constant time	Looking up a single element in an array
$O(\log n)$	Logarithmic	Finding an item in a sorted array with a binary search
$O(n)$	Linear time	Searching an unsorted array for a specific value
$O(n \log n)$	Log-linear	Complex sorting algorithms like heap sort and merge sort
$O(n^2)$	Quadratic	Simple sorting algorithms, such as bubble sort, selection sort, and insertion sort

Introduction to data structures

▼ Data Structures : Arrays

- Collection of elements identified by **index** or key , and it starts with 0

▼ Array



• Array Operations

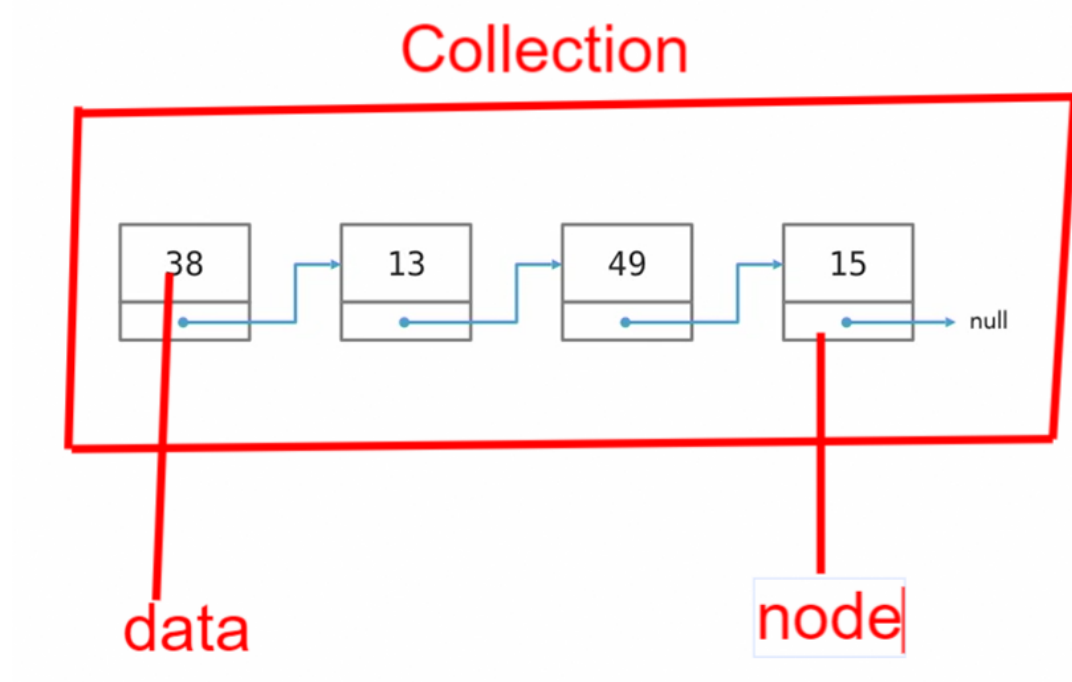
- Calculate item index: $O(1)$
- Insert or delete item at beginning : $O(n)$
- Insert or delete item in middle: (n)
- Insert or delete item at end : $O(1)$

o

▼ Data Structures: **Linked lists**

- is a linear collection of data elements, called nodes
- Contain reference to the next node in the list
- Hold whatever data the application needs

▼ **linked lists**



▼ **advantages**

Linked Lists

- Elements can be easily inserted and removed
- Underlying memory doesn't need to be reorganized
- Can't do constant-time random item access
- Item lookup is linear in time complexity ($O(n)$)

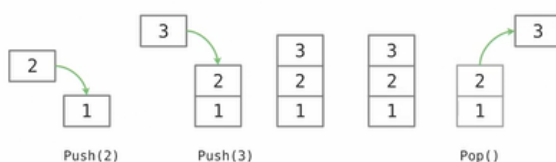
▼ Stacks and Queues

Stack: collection that supports push and pop operations

The last item pushed is the first one popped

- Example
 - Backtracking : browser back stack

▼ pic

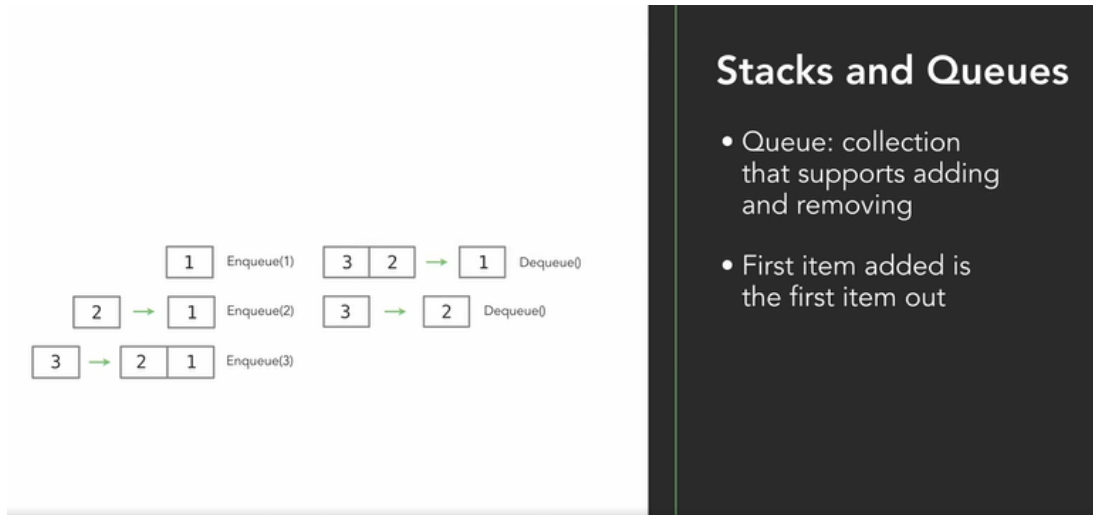


Stacks and Queues

- Stack: collection that supports push and pop operations
- The last item pushed is the first one popped

- Queue: collection that supports adding and removing
- First item added is the first item out

▼ pic#



- Example
 - Messaging

▼ Hash Tables or Dictionary

It create a data (values) , which have specifcs Keys

- Advanteges
 - Key-to-value mappings are unique
 - Hash tables are typically very fast
 - There are for big projects
 - Hash tables don't order entries in a predictable way

```
List = [
```

Recursion

▼ Recursion

- Recursion is when a function calls itself.
- Recursive functions need to have a breaking condition
 - this prevents infinite loops and eventual crashes
- Each time the function is called, the old arguments are saved
 - This is called the “call stack”

▼ Example

Recursion

```
function countdown(x) {  
  if (x == 0)  
    print("done!")  
    return  
  else  
    print(x, "...")  
    countdown(x-1)  
}
```

countdown(4)

Recursive functions
need to have a breaking
condition

Each time the function
is called, the old
arguments are saved

Recursion is when a function calls itself.

Sorting Data

▼ Overview

The bubble sort

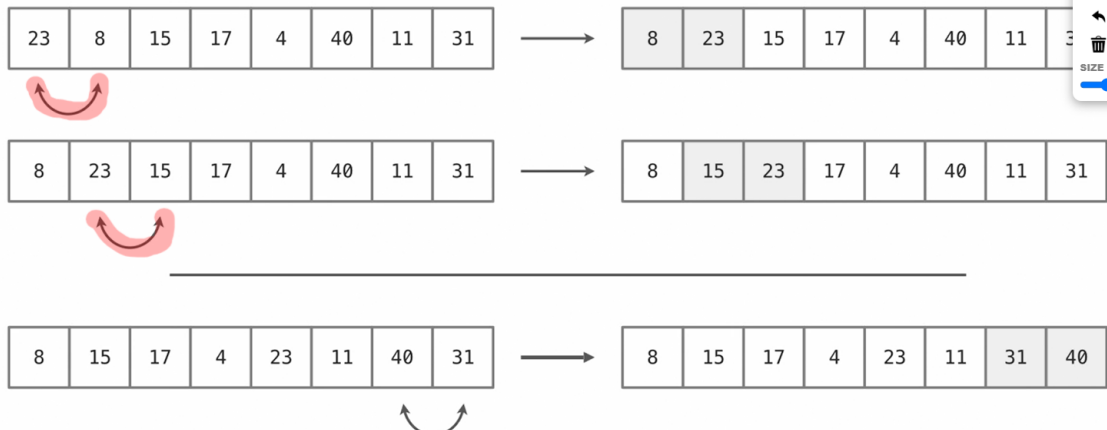
the merge sort

Quicksort

▼ The Bubble Sort

The Bubble Sort

Vergleich die nachbare Nodes und schieb der größere nach recht



▼ The Merge Sort

- Divide - and - conquer algorithm
- Breaks a dataset into individual pieces and merges them
- Uses recursion to operate on datasets
- Performs well on large sets of data
- In general has a performance of $O(n \log n)$ time complexity

▼ Example

```
items = [6, 20, 8, 19, 56, 23, 87, 41, 49, 53, 622]

def mergesort(dataset):
    if len(dataset) > 1:
        mid = len(dataset) // 2
        leftarr = dataset[:mid]
        rightarr = dataset[mid:]

        # TODO: recursively break down the arrays
        mergesort(leftarr)
        mergesort(rightarr)

        # TODO: now perform the merging
        i=0 # index into the left array
        j=0 # index into the right array
        k=0 # index into merged array

        # TODO: while both arrays have content
        while i < len(leftarr) and j < len(rightarr):
            if leftarr[i] < rightarr[j]:
                dataset[k] = leftarr[i]
                i += 1
            else:
```



```

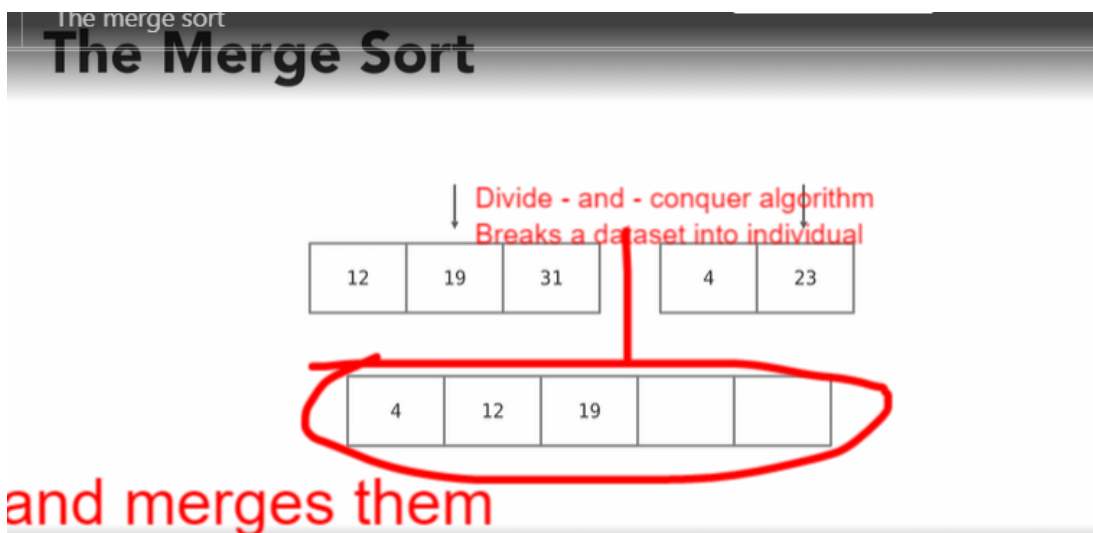
        dataset[k] = rightarr[j]
        j += 1
        k += 1

    # TODO: if the left array still has values, add them
    while i < len(leftarr):
        dataset[k] = leftarr[i]
        i += 1
        k += 1

    # TODO: if the right array still has values, add them

# test the merge sort with data
print(items)
mergesort(items)
print(items)

```



▼ The Quicksort

- Divide-and-conquer algorithm, like the merge sort
- Also uses recursion to perform sorting
- Generally performs better than merge sort, $O(n \log n)$
- Operates in place on the data
- Worst case is $O(n^2)$ when data is mostly sorted already

Searching Data

▼ Ordered list search

- Indexes are created at the beginning and end of the list and a midpoint is calculated. Values are searched between the index and midpoint. The indexes move based on the results.

Correct

Searching an ordered list is efficient. Low and high index values and a midpoint is created in the data set. If the value is not found between an index value and the midpoint, new index values are established. This repeats until the value is found.