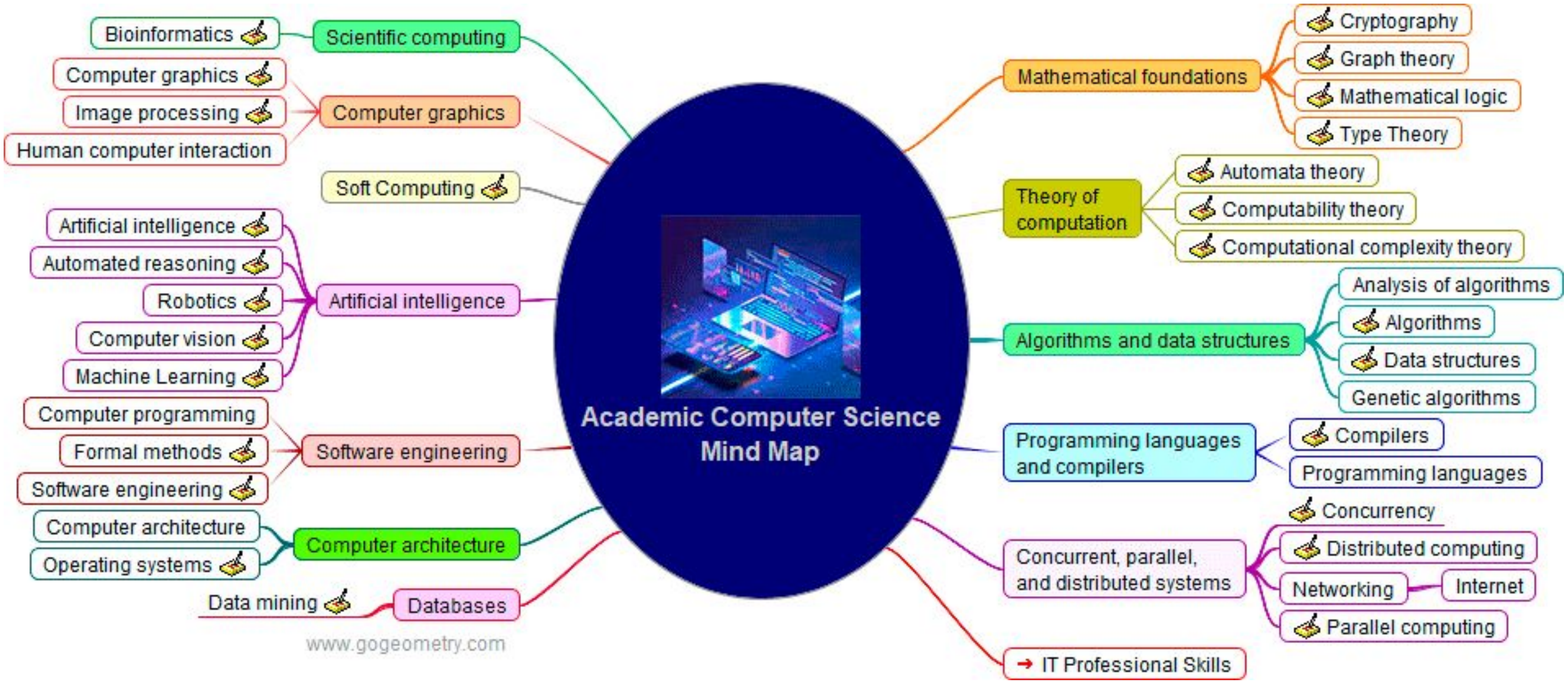
Computer Science

Algorithms Complexity and Data Structures

Computer Science

is the study of computation, information, and automation



Computational Complexity

Time Complexity	Space Complexity
Calculates the time required	Estimates the space (memory) required
Time is counted for all statement	Memory space is counted for all variables, inputs and outputs.
The size of the input data is the primary determinant	Primary determinant is the auxiliary variable size
Deals with the computational time with the change in the size of the input	Deals with how much (extra) space would be required with a change in the input size.

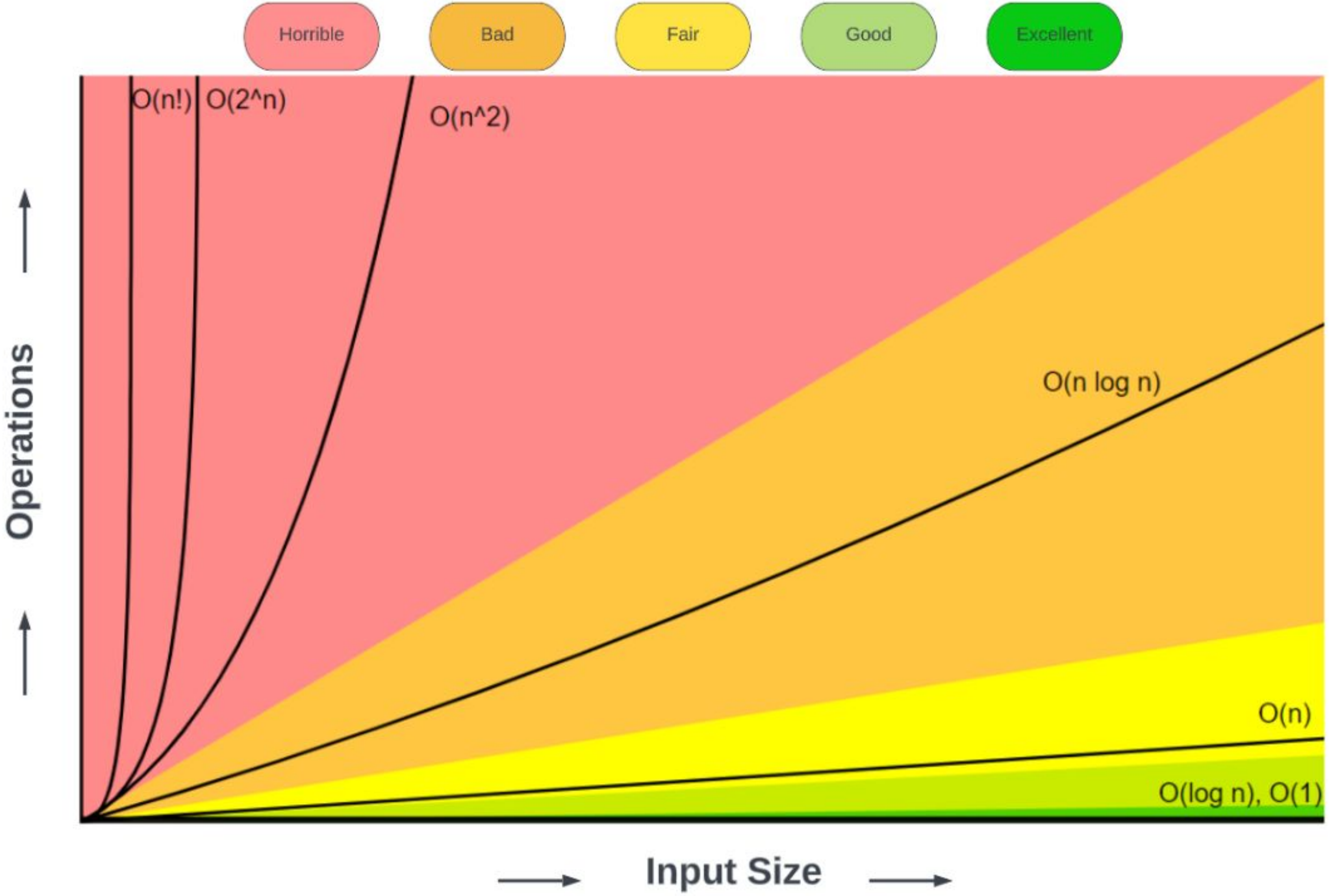
Exact vs Asymptotic Analysis

- exact estimation of execution time
- each base operation has different execution time
- hard to compute

- dependence of the number of operations on input data
- assigning an algorithm to a complexity class
- base operations are “equal”
- easy to estimate

S.No.	Big O	Big Omega (Ω)	Big Theta (Θ)
1.	It is like (<=) rate of growth of an algorithm is less than or equal to a specific value.	It is like (>=) rate of growth is greater than or equal to a specified value.	It is like (==) meaning the rate of growth is equal to a specified value.
2.	The upper bound of algorithm is represented by Big O notation. Only the above function is bounded by Big O. Asymptotic upper bound is given by Big O notation.	The algorithm's lower bound is represented by Omega notation. The asymptotic lower bound is given by Omega notation.	The bounding of function from above and below is represented by theta notation. The exact asymptotic behavior is done by this theta notation.
3.	Big O – Upper Bound	Big Omega (Ω) – Lower Bound	Big Theta (Θ) – Tight Bound
4.	It is define as upper bound and upper bound on an algorithm is the most amount of time required (the worst case performance).	It is define as lower bound and lower bound on an algorithm is the least amount of time required (the most efficient way possible, in other words best case).	It is define as tightest bound and tightest bound is the best of all the worst case times that the algorithm can take.
5.	Mathematically: Big Oh is 0 <= f(n) <= Cg(n) for all n >= n0	Mathematically: Big Omega is 0 <= Cg(n) <= f(n) for all n >= n0	Mathematically – Big Theta is 0 <= C2g(n) <= f(n) <= C1g(n) for n >= n0

Complexity Notations



Time Complexity

Example

```ruby
# O(n) time, O(n) memory
def reverse_map(array)
  array.each_with_index.map { |el, idx| array[- idx - 1] }
end
```

Без использования стандартных функций написать функцию reverse, которая принимает массив и возвращает массив в обратном порядке. Исходный массив мутировать нельзя.

```ruby
# O(n) time, O(n) memory
def reverse_assign(array)
  Array.new(array.size)

  array.size.times do |idx|
    result[- idx - 1] = array[idx]
  end
  result
end
```

```ruby
# O(n²) time, O(n) memory
def reverse_unshift(array)
  result = []

  array.each do |el|
    result.unshift(el)
  end
  result
end
```

```ruby
# O(n²) time, O(n) memory
def reverse_push(array)
  result = []

  (array.size - 1).downto(0) do |idx|
    result << array[idx]
  end
  result
end
```

Algorithms

By execution:

- Linear
- Branching
- Cyclic
- Recursive

By result:

- Deterministic
- Randomized
- Exact
- Heuristic

Sorting Algorithms

- Puts elements of a list into an order
- Stable vs Unstable

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

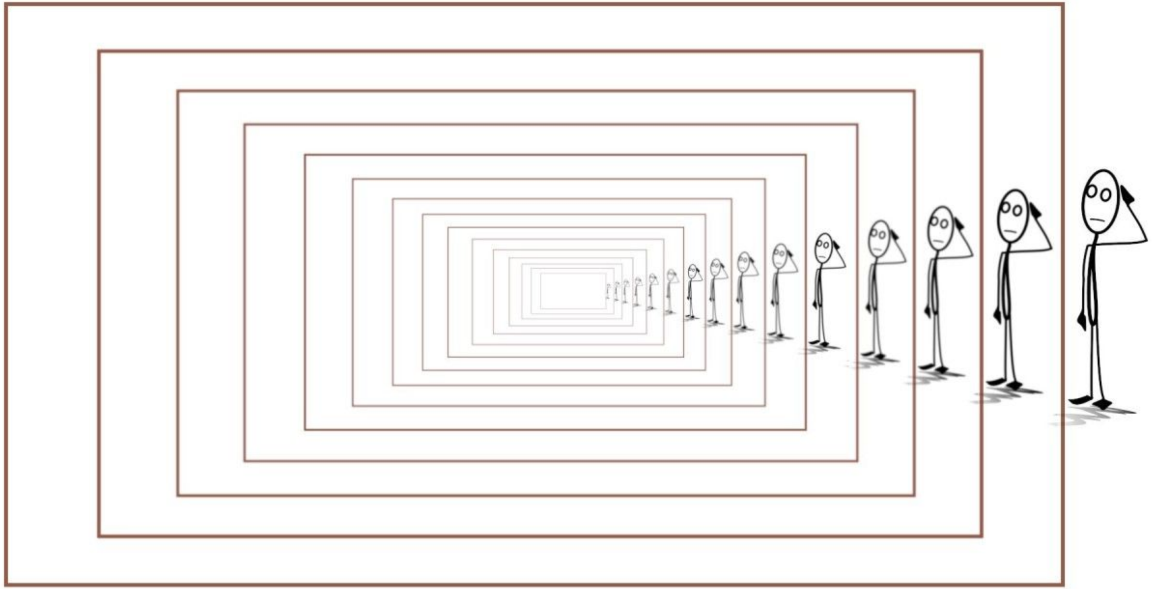
Bubble Sort

```ruby
def bubble_sort(array)
  # Duplicate array to avoid mutation
  sorted  = array.dup

  # Endless loop
  loop do
    # Escape criteria
    swapped = false
    # Push elements one-by-one to the top
    (sorted.size - 1).times do |i|
      # If left element is grater than right swap them
      if sorted[i] > sorted[i + 1]
        sorted[i], sorted[i + 1] = sorted[i + 1], sorted[i]
        # Not ready to escape, need one more run
        swapped = true
      end
    end
    # Escape if not swaps were made
    break unless swapped
  end
  sorted
end
```

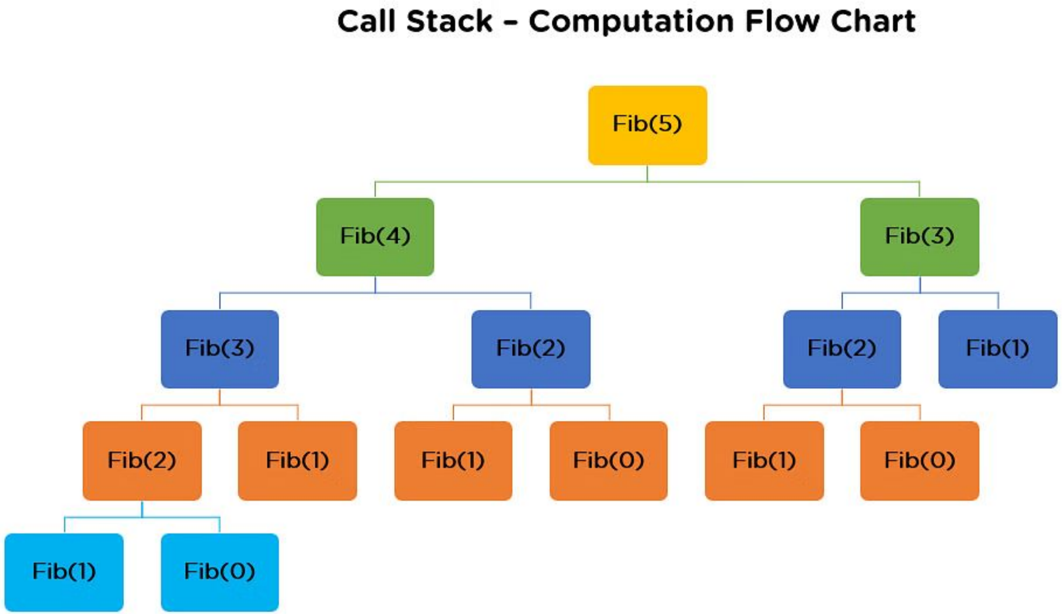
8 5 3 1 4 7 9

Recursive Algorithms

- Function calls itself
- Solve smaller subproblems of the original problems
- Can lead to stack overflow

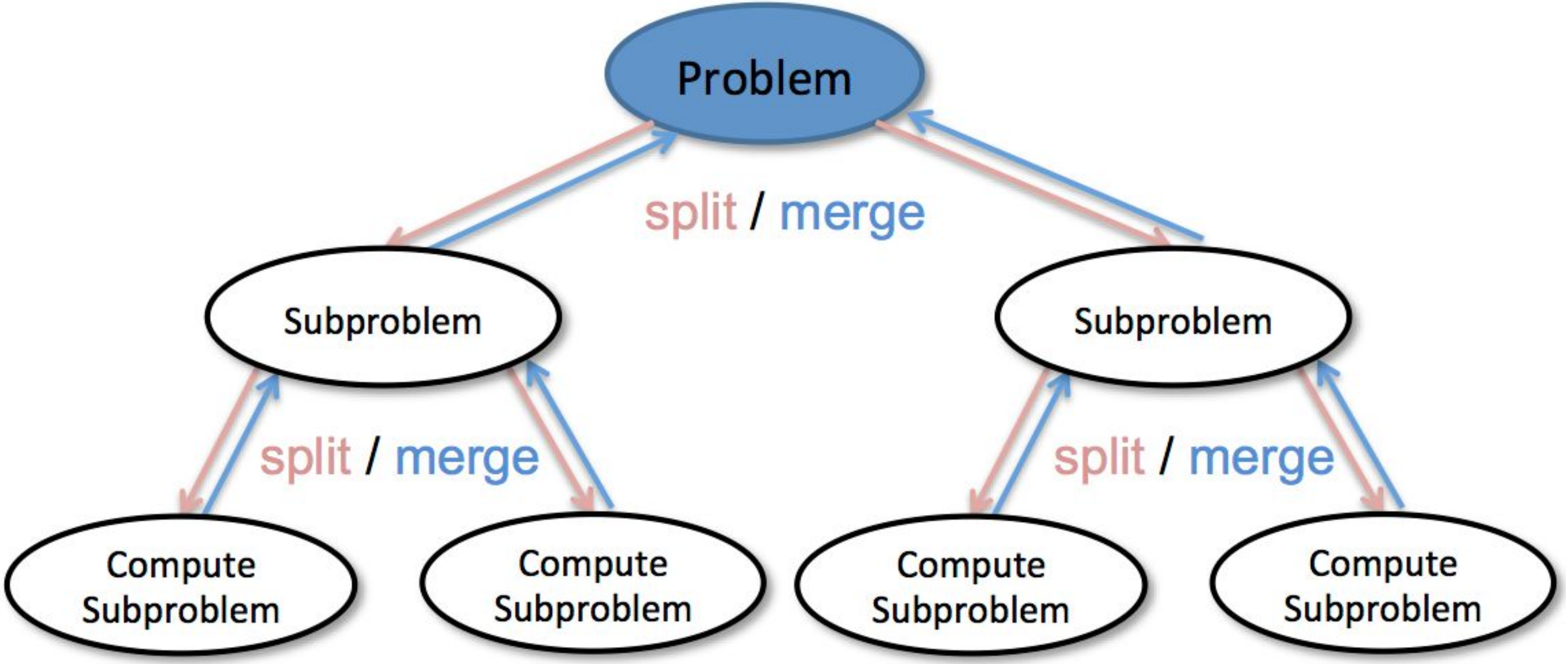


Fibonacci Sequence

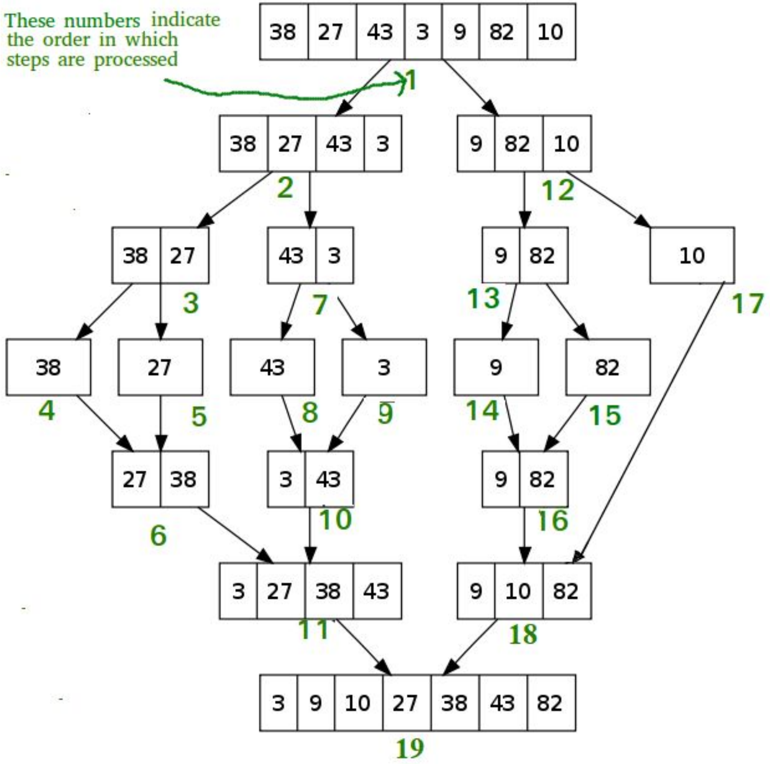



```ruby
def fib(n)
  # Initial values for n=0 and n=1
  return 1 if n < 2

  # Each value is a sum of previous two
  fib(n - 1) + fib(n - 2)
end

p fib(0) # 1
p fib(1) # 1
p fib(2) # 2
p fib(3) # 3
p fib(4) # 5
p fib(5) # 8
p fib(6) # 13
p fib(7) # 21
```

Divide-and-conquer Algorithms



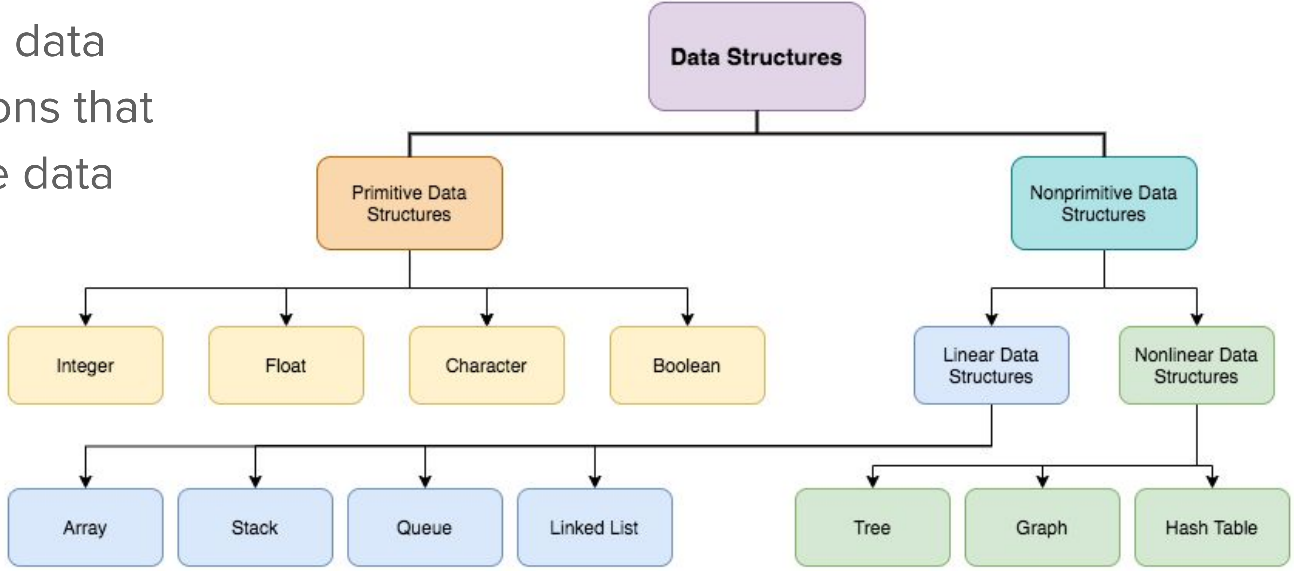
Merge Sort




```ruby
def merge_sort(array)
  # Escape if length is 1 or less
  return array if array.length <=1

  # Find the middle
  mid = array.length/2
  # Take the first half and sort it recursive
  left = merge_sort(array[...mid])
  # Take the second half and sort it recursive
  right = merge_sort(array[mid..])
  # Merge sorted halves
  merge(left, right)
end

def merge(left, right)
  # Predefine array for memory optimization
  sorted = Array.new(left.size + right.size)

  # Set indices to zero
  idx, lidx, ridx = 0, 0, 0
  # Go until our sorted array is full
  while idx < sorted.size  do
    # Escape if one of the arrays ended, fill sorted with the rest
    if left[lidx].nil?
      sorted[idx..] = right[ridx..]
      break
    end
    # Escape if one of the arrays ended, fill sorted with the rest
    if right[ridx].nil?
      sorted[idx..] = left[lidx..]
      break
    end
    # Insert the smallest value to sorted array
    if left[lidx] < right[ridx]
      sorted[idx] = left[lidx]
      lidx += 1
    else
      sorted[idx] = right[ridx]
      ridx += 1
    end
    # Increase index each interation
    idx += 1
  end
  return sorted
end
```

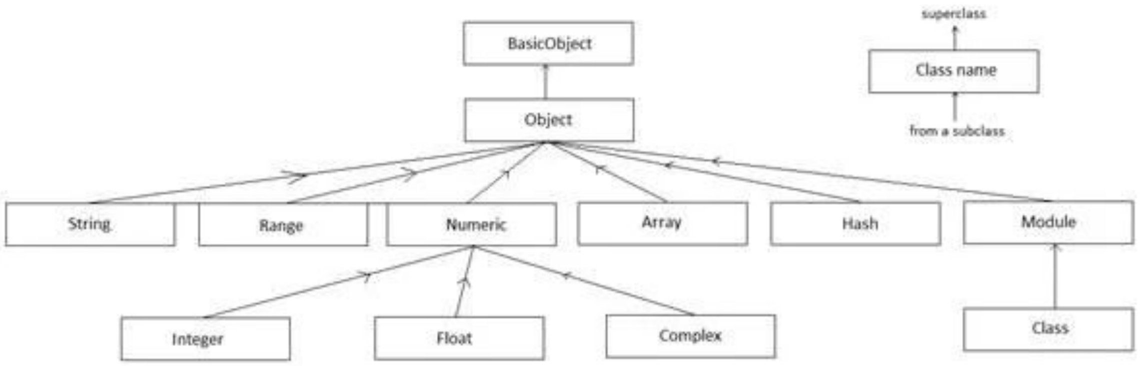
Data Structures

- Data storage and data access
- Relationships among data
- Functions or operations that can be applied to the data



Data Structures in Ruby

- Array
- Hash
- Set
- Implement Enumerable
- Data can be written and read



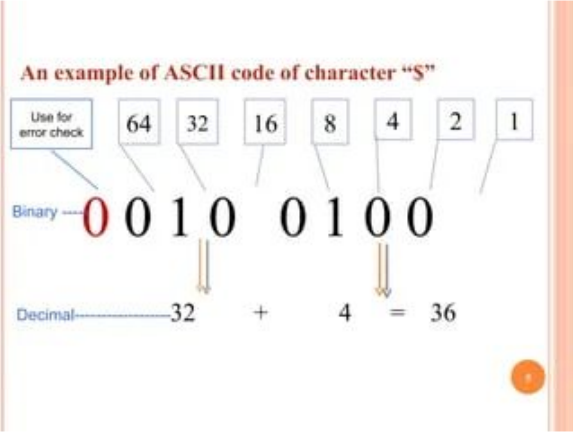
Data Storage

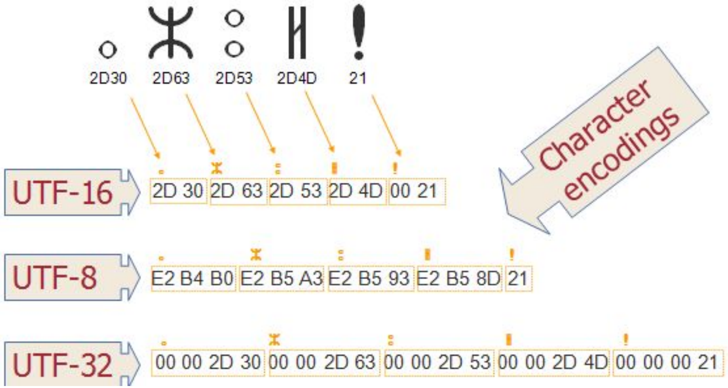
Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647

Character Encoding

ASCII Vs UNICODE

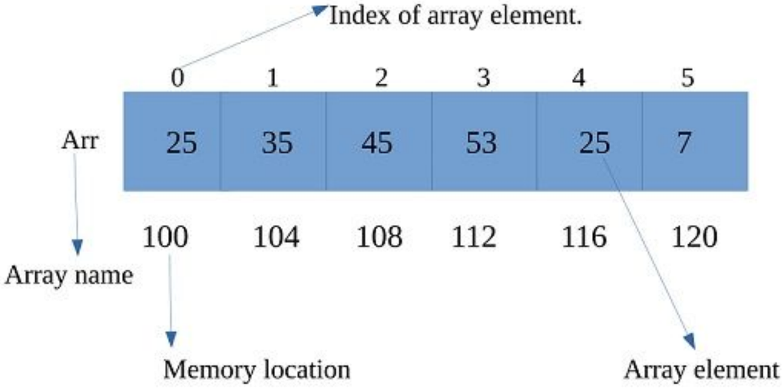
ASCII	UNICODE
A character encoding standard for electronic communication	A computing industry standard for consistent encoding, representation, and handling of text expressed in most of the world's writing systems
Stands for American Standard Code for Information Interchange	Stands for Universal Character Encoding
Supports 128 characters	Supports a wide range of Character set
Uses 7 bits to represent a character	Uses 8 bit, 16 bit or 32 bit depending on the encoding type
Requires less space	Requires more space





(Dynamic) Array

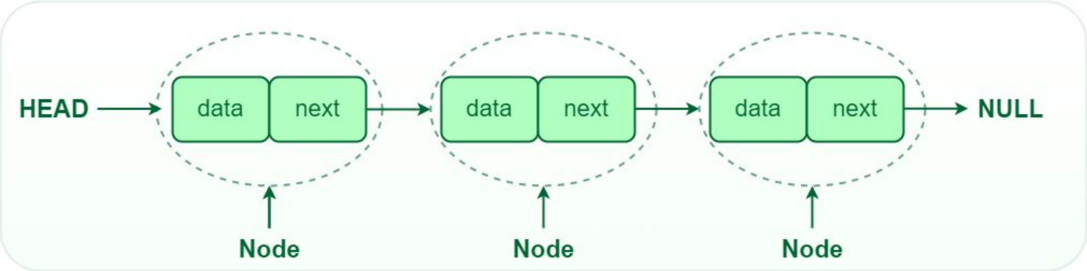
- Collection of values
- Ordered
- Represented as a pointer to the beginning, size and type
- Dynamically resized when needed



Index	**Mutation (beginning)**	**Mutation (middle)**	**Mutation (end)**
O(1)	*O(n)*	*O(n)*	*O(1) amortized*

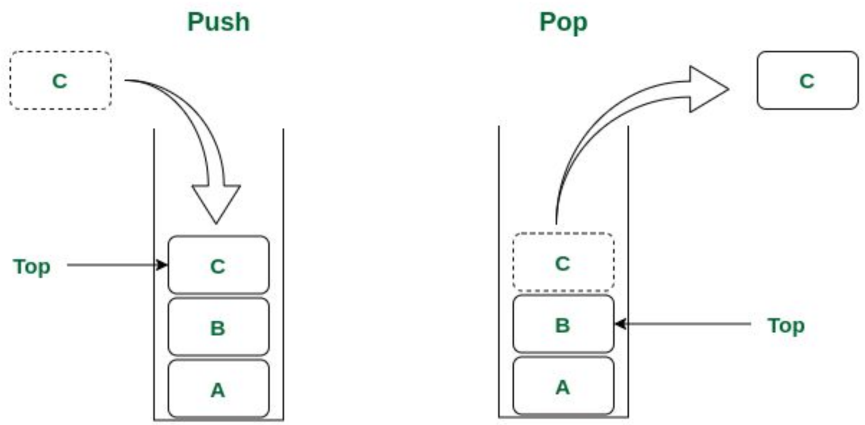
Linked List

- Collection of values
- Ordered
- Represented as a node with a link to the next element
- Not bound to size



Index	Mutation (beginning)	Mutation (middle)	Mutation (end)
O(n)	*O(1)*	*O(n) - unknown* *O(1) - known*	*O(n) - unknown* *O(1) - known*

Stack and Queue

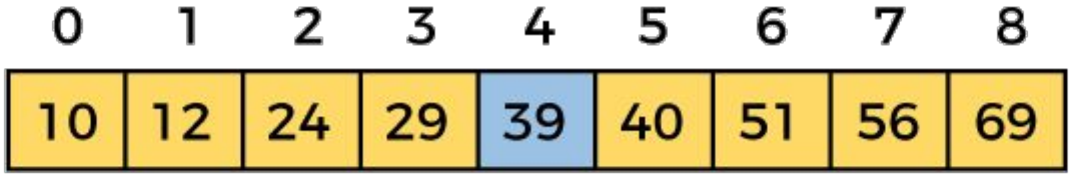


Stack Data Structure

Queue Data Structure

Search Algorithms

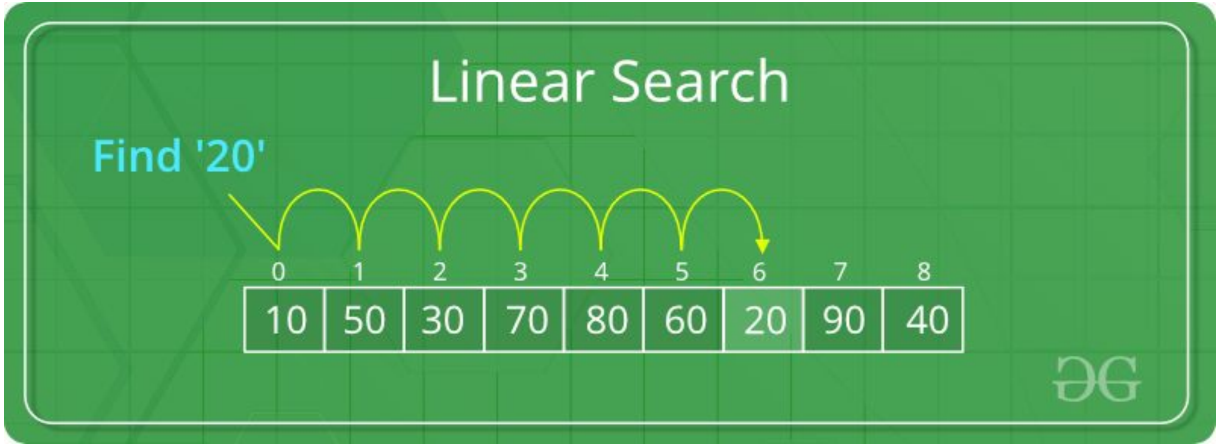
- Find min/max value
- Find index/node with value
- Find the nearest value



Linear Search

- Can be applied to any data structures
- Complexity ***O(n)***

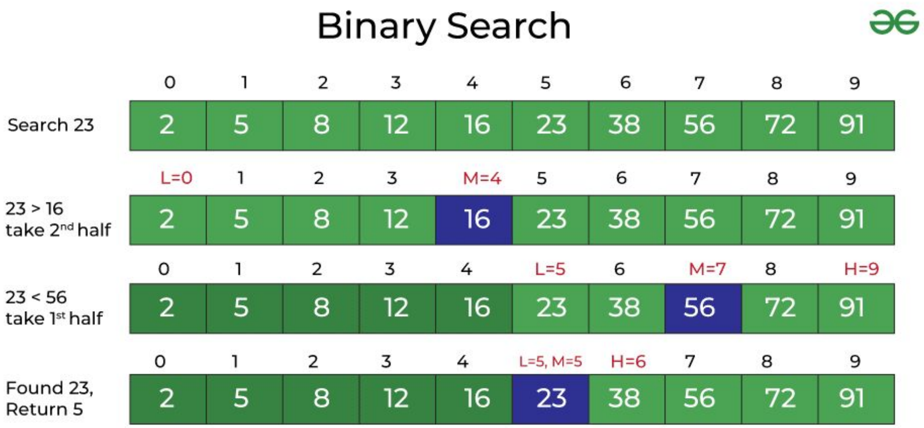
```ruby
class Array
  def linear_search(val)
    each_with_index do |el, idx|
      return idx if val == el
    end
  end
end

puts [1, 2, 3, 4, 5, 6].linear_search(3) # 2
```



Binary Search

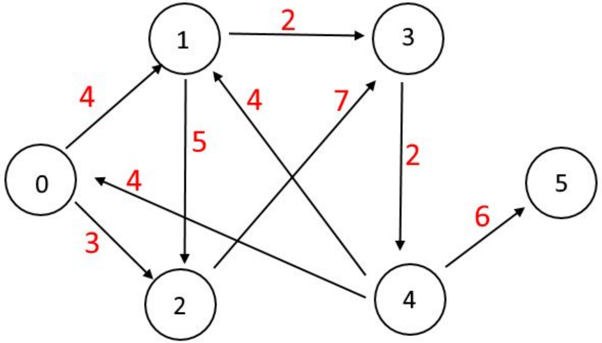
- Can be applied only to sorted structures
- Complexity ***O(log n)***

```ruby
  def binary_search(val, idx = 0)
    # If size is 1 we either found our value or it doesn't exist
    return (val == first ? idx : nil) if size == 1

    middle = size / 2
    # If middle value is less than our value
    # then we search in the second half
    if val < self[middle]
      self[...middle].binary_search(val, idx)
    # If middle value is greater than or equal our value
    # then we search in the first half
    else
      self[middle..].binary_search(val, idx + middle)
    end
  end
end

puts [1, 2, 3, 4, 5, 6 ,7, 8].binary_search(4) # 3
puts [1, 2, 3, 4, 5, 6 ,7, 8].binary_search(13) # nil
```

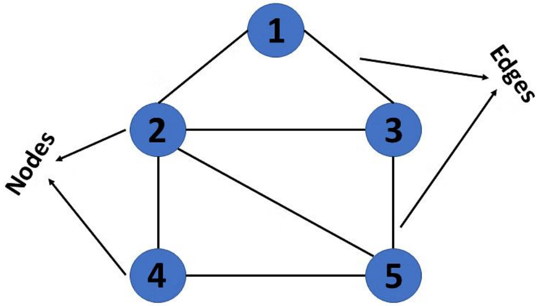


Graph

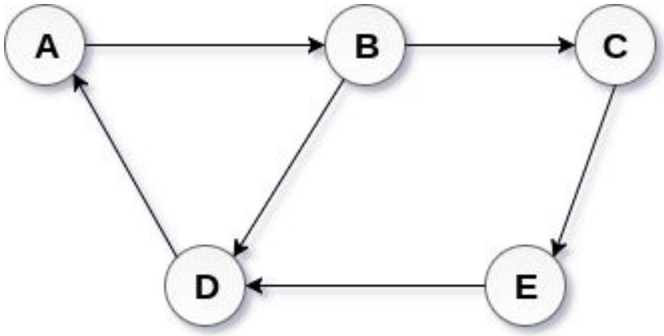
- Nonlinear
- Collection of vertices (nodes) and edges
- Represented as a pointer to the root node



Directed Weighted Graph



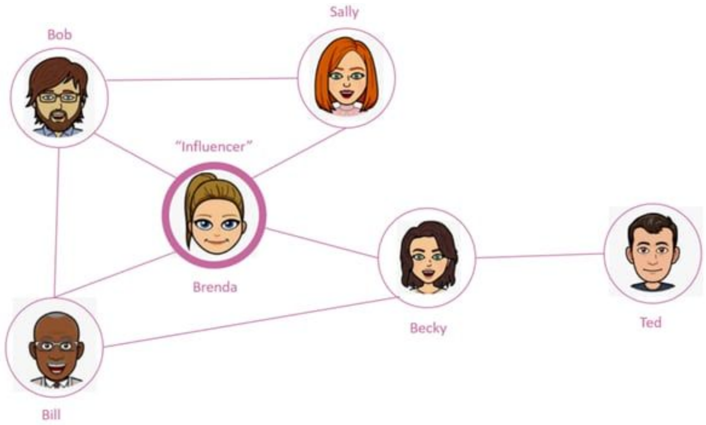
Graph

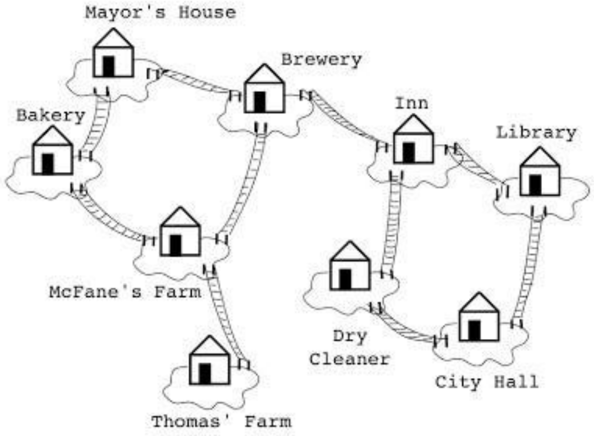


Directed Graph

Graph

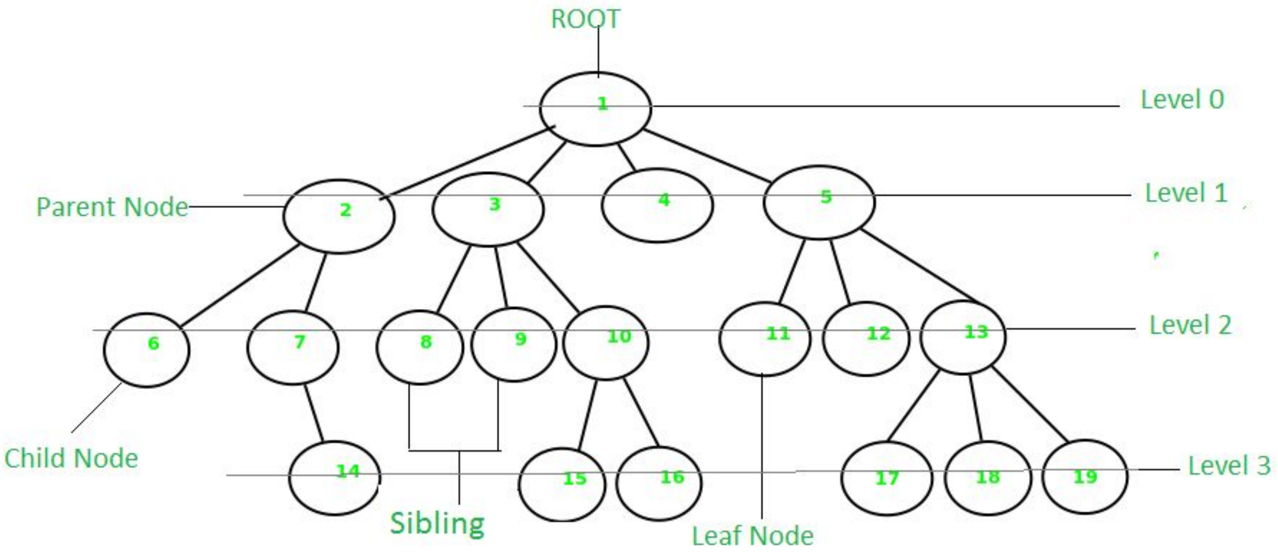
- Social Network
- Set of locations with transitions (railroad, airports)





Tree

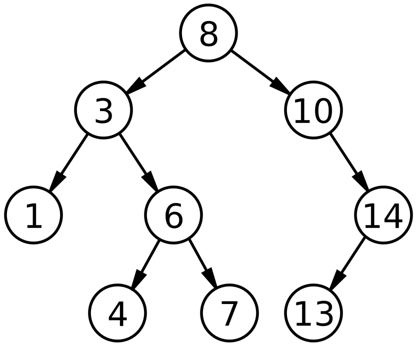
- Graph without cycles
- Collection of vertices (nodes) and edges
- Always has root and leaves
- Has levels (generations) depending on a root selection



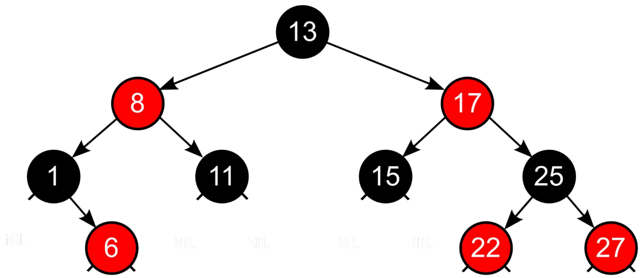
Tree

- Binary
    - Binary Heap
    - Binary Search Tree
- Self-balancing
    - Red–Black Tree
    - B-tree

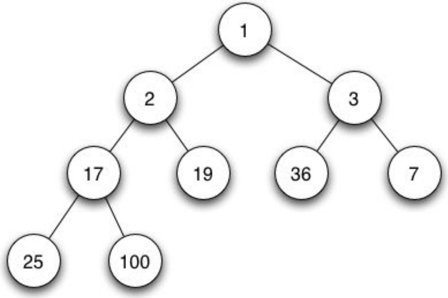
Search	Insertion	Deletion
O(log n)	*O(log n)*	*O(log n)*



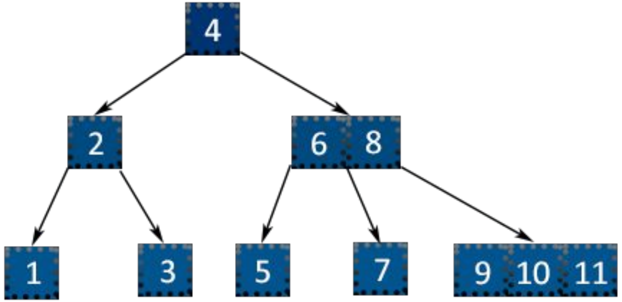
Binary Search Tree



Red-Black Tree



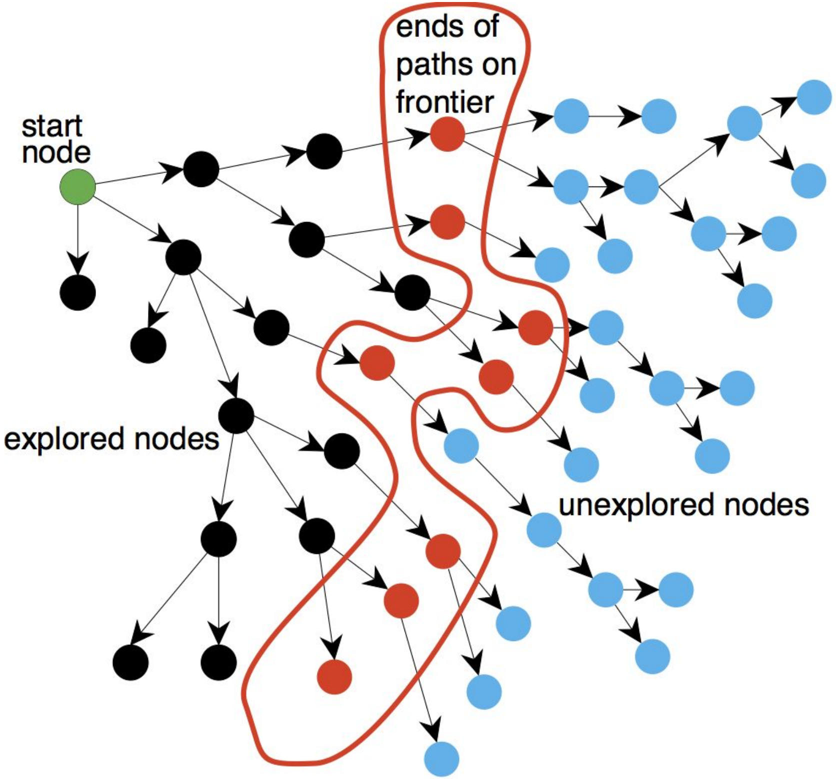
Binary Heap



B-tree

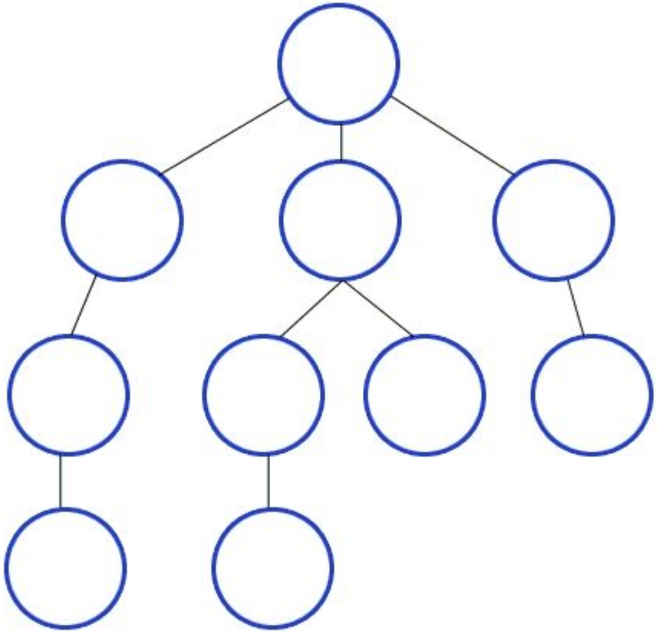
Graph Search

- Done by visiting each vertex
- Can be done depth-first or breadth-first
- Can be implemented recurrently or using queue/stack



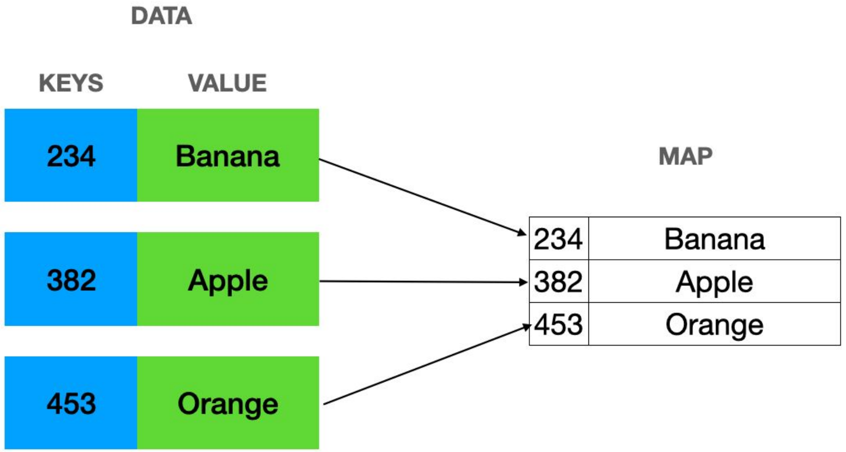
Depth-First Search

```ruby
def dfs(skey)
  puts self
  return self.slice(skey) if key?(skey)

  keys.each do |key|
    if self[key].is_a?(Hash)
      result = self[key].dfs(skey)
      return result if result
    end
  end
  nil
end
```



Map or Associative Array or Dictionary

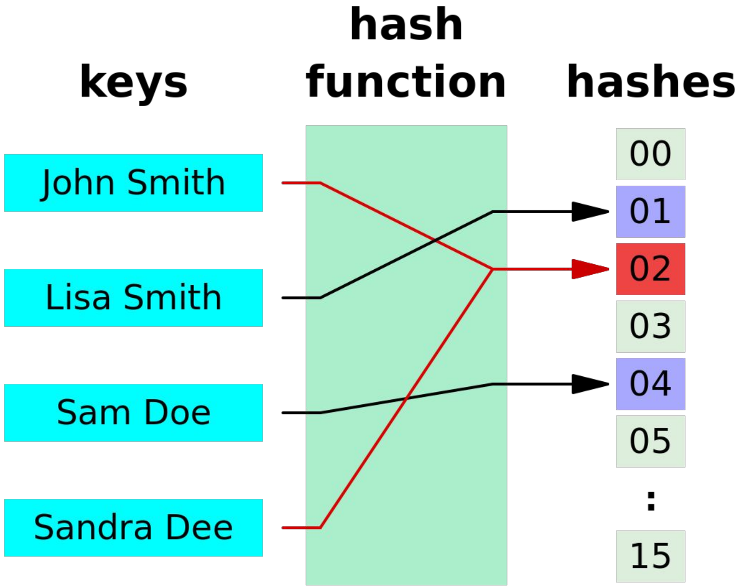
- Nonlinear
- Collection of key-value pairs
- Implemented as a ***Hash Table*** or as a ***Search Tree***



Search	Insertion	Deletion
O(log n)	*O(log n)*	*O(log n)*

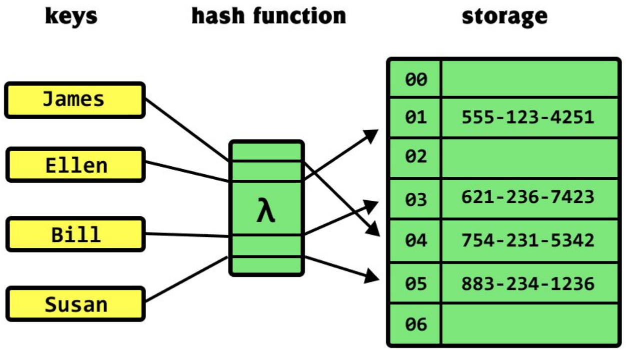
Hash Function

- Transforms input to hash value of fixed length
- Resulting values are uniformly distributed over the keyspace
- Should be very fast to compute
- Should minimize duplication of output values (collisions)



Hash Map

- Nonlinear
- Collection of key-value pairs
- Unordered
- Stored as an array



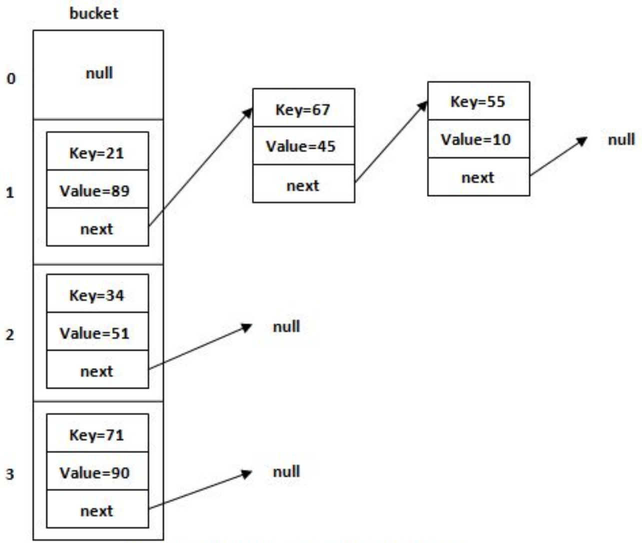


Figure: Allocation of nodes in Bucket

Search	Insertion	Deletion
O(1)	*O(1)*	*O(1)*

Application in QA

- Each test case is an algorithm
- Performance testing
- Autotests memory and time optimization

Thank you for your attention!