Algorithms and Data Structures (ALG)

Indhold

[Concepts 2](#_Toc41515108)

[Computational Problem 2](#_Toc41515109)

[Algorithm and Pseudocode 2](#_Toc41515110)

[Worst-case and best-case analysis 2](#_Toc41515111)

[Asymptotic Notation (big-O, big-Omega, and big-Theta) 3](#_Toc41515112)

[Properties of Asymptotic Notations : 5](#_Toc41515113)

[Proving the correctness of an algorithm (using e.g., induction or loop invariants) 6](#_Toc41515114)

[Solving Recurrences (using, substitution, recursion-tree, and Master methods) 8](#_Toc41515115)

[max/min-heap property 11](#_Toc41515116)

[Hash functions and their properties 13](#_Toc41515117)

[Binary search tree property 15](#_Toc41515118)

[Red-black-tree property 15](#_Toc41515119)

[Optimal Substructure Property 15](#_Toc41515120)

[Graph of dependencies 15](#_Toc41515121)

[Graphs, cycles on graphs, negative cycles, directed acyclic graphs 15](#_Toc41515122)

[Topological ordering on an acyclic graph 15](#_Toc41515123)

[shortest paths problems on graphs and weighted graphs 15](#_Toc41515124)

[Algorithms 16](#_Toc41515125)

[Insertion Sort 16](#_Toc41515126)

[Merge Sort 16](#_Toc41515127)

[Heapsort 16](#_Toc41515128)

[Quicksort 16](#_Toc41515129)

[Counting Sort 16](#_Toc41515130)

[Radix Sort 16](#_Toc41515131)

[Breadth-First Search 16](#_Toc41515132)

[Depth-first Search 16](#_Toc41515133)

[Topological Sorting 16](#_Toc41515134)

[Computing Strongly Connected Components 16](#_Toc41515135)

[Bellman-Ford Algorithm 16](#_Toc41515136)

[Dijkstra’s algorithm 16](#_Toc41515137)

[Floyd-Warshall algorithm 16](#_Toc41515138)

[Transitive closure of a directed graph 16](#_Toc41515139)

[Data Structures 16](#_Toc41515140)

[Max-Heaps & Min-Heaps 16](#_Toc41515141)

[Stacks 16](#_Toc41515142)

[Queues 16](#_Toc41515143)

[Linked Lists (both doubly-linked and singly-linked) 16](#_Toc41515144)

[Rooted Trees 16](#_Toc41515145)

[Hash Tables (direct-address, open addressing) 16](#_Toc41515146)

[Binary Search Trees 16](#_Toc41515147)

[Red-Black Trees 16](#_Toc41515148)

[Directed and Undirected Graphs (adjacency-list & adjacency-matrix representations) 16](#_Toc41515149)

[Weighted Graphs (adjacency-list & adjacency-matrix representations) 16](#_Toc41515150)

[Algorithm Design 16](#_Toc41515151)

[Divide and Conquer 16](#_Toc41515152)

[Dynamic Programming 16](#_Toc41515153)

## Concepts

### Computational Problem

A computational problem can be viewed as an infinite collection of instances together with a solution for every instance.

### Algorithm and Pseudocode

### Worst-case and best-case analysis

Worst case:  
In the worst case analysis, we calculate upper bound - O(f(n)) on running time of an algorithm.

We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched for is not present in the array. The function will then compare it with all the elements of arr[] one by one. And find nothing going through the whole array.

#### Best case:

In the best case analysis, we calculate lower bound - Ω(f(n)) on running time of an algorithm.  
We must know the case that causes minimum number of operations to be executed.   
**Example:** In the linear search problem, the best case occurs when x is present at the first location.   
The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1)

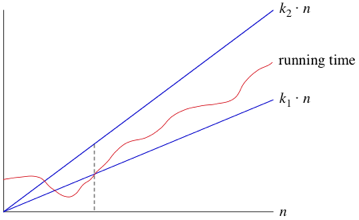
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | Data structure | Time complexity:Best | Time complexity:Average | Time complexity:Worst | Space complexity:Worst |
| Quick sort | Array | O(*n* log(*n*)) | O(*n* log(*n*)) | O(*n*2) | O(*n*) |
| Merge sort | Array | O(*n* log(*n*)) | O(*n* log(*n*)) | O(*n* log(*n*)) | O(*n*) |
| Heap sort | Array | O(*n* log(*n*)) | O(*n* log(*n*)) | O(*n* log(*n*)) | O(1) |
| Smooth sort | Array | O(*n*) | O(*n* log(*n*)) | O(*n* log(*n*)) | O(1) |
| Bubble sort | Array | O(*n*) | O(*n*2) | O(*n*2) | O(1) |
| Insertion sort | Array | O(*n*) | O(*n*2) | O(*n*2) | O(1) |
| Selection sort | Array | O(*n*2) | O(*n*2) | O(*n*2) | O(1) |
| Bogo sort | Array | O(*n*) | O(*n* *n*!) | O(∞) | O(1) |

### Asymptotic Notation (big-O, big-Omega, and big-Theta)

For each little computation it takes a constant amount of time each time it executes. If the for-loop iterates  times, then the time for all  iterations is c1 ​⋅, is the sum of the times for the computations in one loop iteration.

#### Big-Theta - Θ(n) (asymptotically tight bound)

When we say that a particular running time is Θ(n) , we're saying that once ​ gets large enough, the running time is at least c1 ​⋅ and at most c2 ​⋅ for some constants c1 and c2:

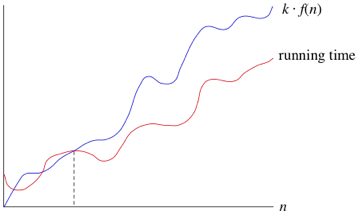


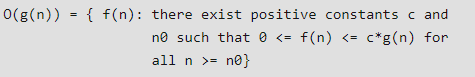
As long as these constants c1 and c2 exist, we say that the running time is Θ(n).

When you use big-Θ notation, you remove the constants and can therefore just say Θ(n).



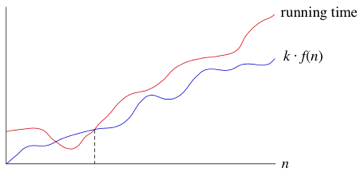
big-O - O(f(n)) (asymptotic upper bounds)  
  
“the running time grows at most this much, but it could grow more slowly."

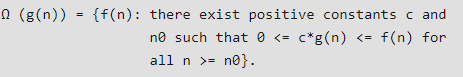
If a running time is O(f(n)), then for large enough  , the running time is at most , for some constant c:  




#### big-Omega - Ω(f(n)) (asymptotic lower bounds)

If a running time is Ω(f(n)) then for large enough , the running time is at least  for some constant c:





Properties of Asymptotic Notations :  
As we have gone through the definition of this three notations let’s now discuss some important properties of those notations.

#### General Properties :

If f(n) is O(g(n)) then a\*f(n) is also O(g(n)) ; where a is a constant.

**Example**: f(n) = 2n²+5 is O(n²)  
**then** 7\*f(n) = 7(2n²+5)  
= 14n²+35 is also O(n²)

Similarly this property satisfies for both Θ and Ω notation.  
We can say  
If f(n) is Θ(g(n)) then a\*f(n) is also Θ(g(n)) ; where a is a constant.  
If f(n) is Ω (g(n)) then a\*f(n) is also Ω (g(n)) ; where a is a constant.

#### Reflexive Properties :

If f(n) is given then f(n) is O(f(n)).

**Example**: f(n) = n² ; O(n²) i.e O(f(n))

Similarly this property satisfies for both Θ and Ω notation.  
We can say  
If f(n) is given then f(n) is Θ(f(n)).  
If f(n) is given then f(n) is Ω (f(n)).

#### Transitive Properties :

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)) .

**Example**: if f(n) = n , g(n) = n² and h(n)=n³  
n is O(n²) and n² is O(n³)  
**then** n is O(n³)

Similarly this property satisfies for both Θ and Ω notation.  
We can say  
If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .  
If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

#### Symmetric Properties :

If f(n) is Θ(g(n)) then g(n) is Θ(f(n)) .

**Example**: f(n) = n² and g(n) = n²  
**then** f(n) = Θ(n²) and g(n) = Θ(n²)

**This property only satisfies for Θ notation.**

Transpose Symmetric Properties :

If f(n) is O(g(n)) then g(n) is Ω (f(n)).

**Example**: f(n) = n , g(n) = n²  
**then** n is O(n²) and n² is Ω (n)

**This property only satisfies for O and Ω notations**.

#### Some More Properties :

If f(n) = O(g(n)) and f(n) = Ω(g(n)) then f(n) = Θ(g(n))

If f(n) = O(g(n)) and d(n)=O(e(n))  
**then** f(n) + d(n) = O( max( g(n), e(n) ))  
**Example**: f(n) = n i.e O(n)  
d(n) = n² i.e O(n²)  
**then** f(n) + d(n) = n + n² i.e O(n²)

If f(n)=O(g(n)) and d(n)=O(e(n))  
**then** f(n) \* d(n) = O( g(n) \* e(n) )  
**Example**: f(n) = n i.e O(n)  
d(n) = n² i.e O(n²)  
then f(n) \* d(n) = n \* n² = n³ i.e O(n³)

### Proving the correctness of an algorithm (using e.g., induction or loop invariants)

Induction  
mathematical demonstration of the validity of a law concerning all the positive integers by proving that it holds for the integer 1 and that if it holds for an arbitrarily chosen positive integer k, it must hold for the integer k + 1.

Step 1. Show it is true for the **first one**

Step 2. Show that if **any one** is true then the **next one** is true

**Example:** is 3n−1 a multiple of 2?

**1.** Show it is true for **n=1**

Yes 2 is a multiple of 2. That was easy.

**2.** Assume it is true for **n=k**

is true

(Not necessarly true but, it is an assumption ... that we treat as a fact for the rest of this example)

Now, prove that  is a multiple of 2

 is also

is also +

And each of these are multiples of 2

Because:

 is a multiple of 2 (we are multiplying by 2)

is true (we said that in the assumption above)

**Example:** Show that 2n < (n + 1)!, for n ≥ 2.

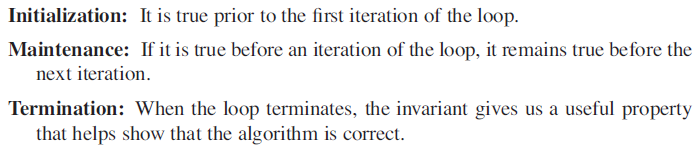
**Proof**: The base step is as follows:  
**Base Step**: n = 2: we have to check that 22 < 3!, but 4 < 6, so this is true.   
**Inductive Step:** Assume 2k < (k + 1)!, for k ≥ 2. We have  
2k+1 = 2 · 2k < 2 · (k + 1)!   
Since k ≥ 2, we have (k + 2) > 2. Therefore:  
2 · (k + 1)! < (k + 2) · (k + 1)! = (k + 2)!  
This completes the inductive step.

#### Loop invariants

An inductive statement which says something which is always true about a program loop.  
Typically written as: S(n) where n is some changing element of the loop:

* The loop counter / Index
* A loop variable which changes on each iteration

To prove a loop invariant you to prove 3 things:



**Example:** Pre-condition: m nonnegative integer, x is a real number, i = 0, and exp = 1.  
**Program:**   
While (I ≠ m)

1. Exp = exp\*x
2. I = I +1

End while

Post-condition: exp = xm

Loop-invariant: S(n) is i = n and exp = xn

**Proof:**

**Basis property**:   
n = 0, then i = 0, exp = x0 = 1, cause exp started on 1.

**Inductive property:**  
I ≠ m and S(k), then:

By 1: expnew = expold \* x = xk \* x = xk +1

By 2: inew = iold + 1 = k + 1

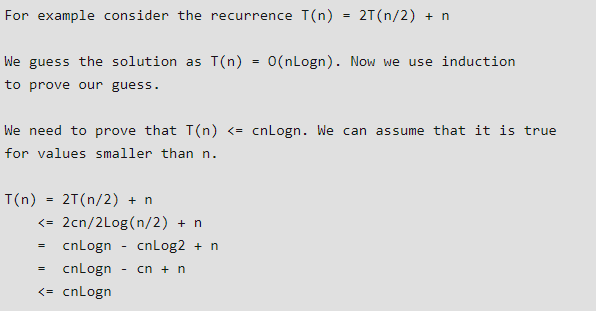
**Eventaul falsity of the Guard:**   
At each iteration, i = i + 1, and i = 0 at the start, so after m iterations we have i = m.

Correctness of the Post-Condition: Guard false implies i = m after m iterations and S(m) = xm

### Solving Recurrences (using, substitution, recursion-tree, and Master methods)

#### Substitution

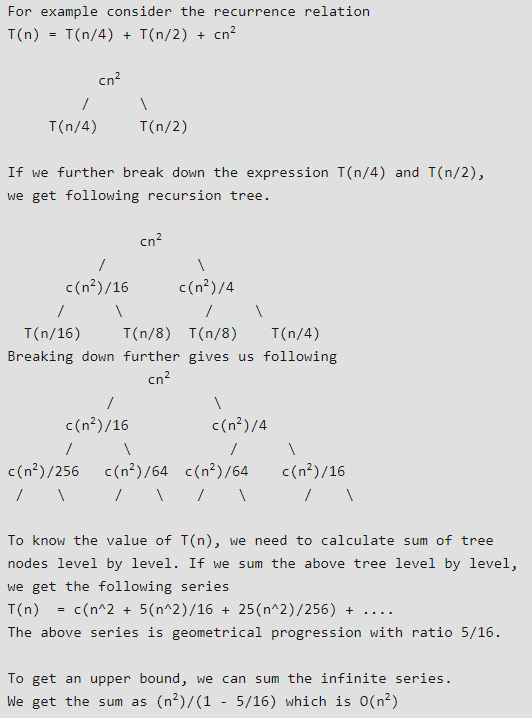
We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.



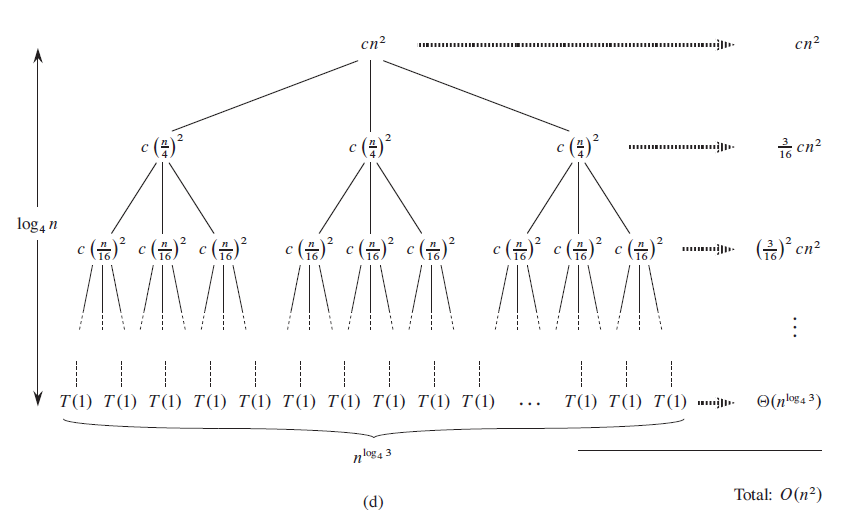
(se kapitel 4.3)

#### Recursion-Tree

In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.



Recursion-trees bliver brugt til at lave et gæt til substitutions metoden. Her tegner man et træ hvor noderne repræsentere et del-problem (hvor ”dyrt” er et del-problem at løse). Vi summere cost i vært niveau af træet og summere vært niveau for at få den totale værdi.



Figuren viser recurrence: T(n) = 3T (n/4) + cn^2

Denne metode er bedst brugt til at finde et gæt til at godkende med substitutions metoden.

#### Master

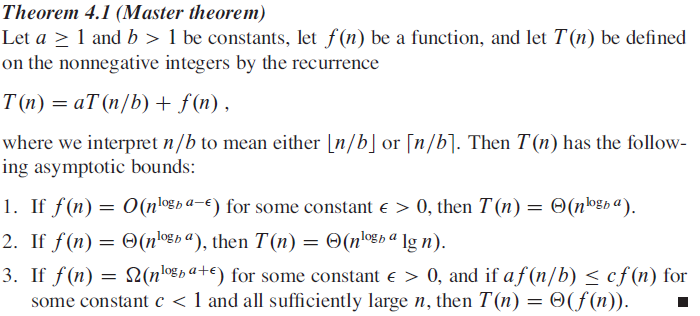
Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.



The three cases:

1. If f(n) = Θ(nc) where c < Logba then T(n) = Θ(nLogba)
2. If f(n) = Θ(nc) where c = Logba then T(n) = Θ(ncLog n)
3. If f(n) = Θ(nc) where c > Logba then T(n) = Θ(f(n))

Master metoden afhænger af følgende sætning:



### max/min-heap property

#### Min heap

In a min heap the parent node is always smaller than or equal to its child node.

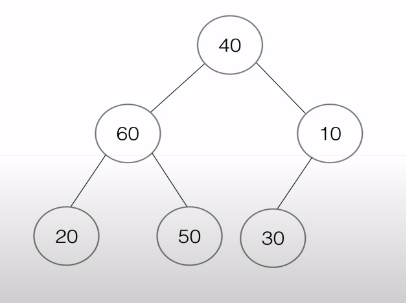
We can represent a min heap using one diemensional array

If N denotes the indes of a parent node then 2N denotes the left child node and 2N +1 denotes the right child node where N = 1,2,3…

**Example:** Input 40 60 10 20 50 30  
There are 6 elements so our heap will have 6 nodes  
We can represent the nodes of the heap in an array

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 40 | 60 | 10 | 20 | 50 | 30 |

We then build the heap



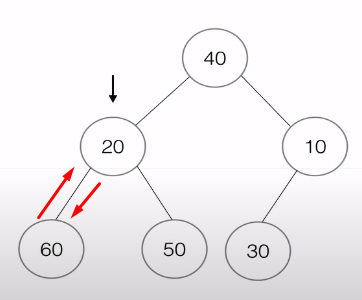
Then we transform it into an min heap

Note! - If there are N nodes then we start comparison from floor(N/2) index.  
**Example:** If N = 5   
Then floor(N/2)  
= floor(5/2) = floor(2.5) = 2 We lower down the value to nearest integer.

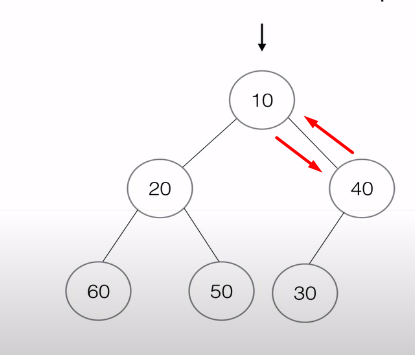
In this case there are 6 nodes, N = 6 so floor(6/2) = floor (3) = 3.   
We will start comparison from index 3 or the 3rd node.

Which is 10, we then ask ourselves is there any child node smaller than 10 ? No  
So we move to index 2 or the 2nd node.

Is there any node smaller than 60 ? Yes so we swap position of 20 and 60

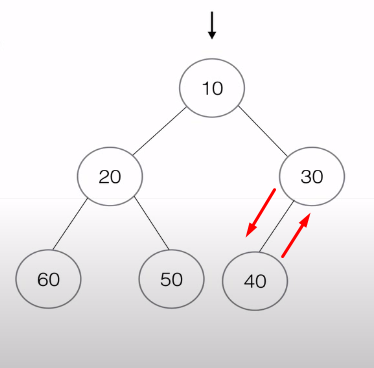


We then move to the 1st node. And swap 10 and 40.



Now that we have reached the 1st node we will check wheter we have created a mini heap

Lastly we swap 30 and 40.



And the array will now look like

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 10 | 20 | 30 | 60 | 50 | 40 |

#### Max heap

In a max heap the parent node is always great than or equal to its child node

We can represent a max heap using one dimensional array

If N denotes the indes of a parent node then 2N denotes the left child node and 2N+1 denotes the right child node where N = 1,2,3…

Samme princip som før, start comparison from floor(N/2). Og swap vis der er et child node der er større end parent node.

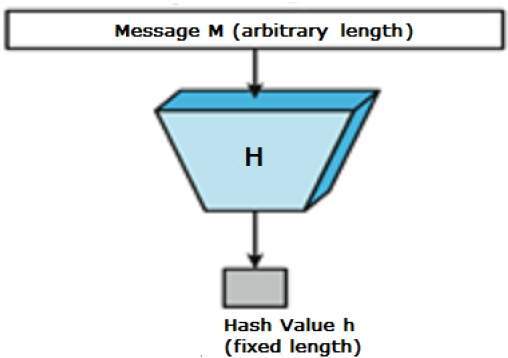
Array heap ouput vil være:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 60 | 50 | 30 | 20 | 40 | 10 |

### Hash functions and their properties

Hash function is a mathematical function that converts a numerical input value into another compressed numerical value. The input to the hash function is of arbitrary length but output is always of fixed length.

Values returned by a hash function are called message digest or simply hash values. The following picture illustrated hash function:



#### Features of Hash Functions

The typical features of hash functions are −

* **Fixed Length Output (Hash Value)**
  + Hash function coverts data of arbitrary length to a fixed length. This process is often referred to as **hashing the data**.
  + In general, the hash is much smaller than the input data, hence hash functions are sometimes called **compression functions**.
  + Since a hash is a smaller representation of a larger data, it is also referred to as a **digest**.
  + Hash function with n bit output is referred to as an **n-bit hash function**. Popular hash functions generate values between 160 and 512 bits.
* **Efficiency of Operation**
  + Generally for any hash function h with input x, computation of h(x) is a fast operation.
  + Computationally hash functions are much faster than a symmetric encryption.

#### Properties of Hash Functions

In order to be an effective cryptographic tool, the hash function is desired to possess following properties −

* **Pre-Image Resistance**
  + This property means that it should be computationally hard to reverse a hash function.
  + In other words, if a hash function h produced a hash value z, then it should be a difficult process to find any input value x that hashes to z.
  + This property protects against an attacker who only has a hash value and is trying to find the input.
* **Second Pre-Image Resistance**
  + This property means given an input and its hash, it should be hard to find a different input with the same hash.
  + In other words, if a hash function h for an input x produces hash value h(x), then it should be difficult to find any other input value y such that h(y) = h(x).
  + This property of hash function protects against an attacker who has an input value and its hash, and wants to substitute different value as legitimate value in place of original input value.
* **Collision Resistance**
  + This property means it should be hard to find two different inputs of any length that result in the same hash. This property is also referred to as collision free hash function.
  + In other words, for a hash function h, it is hard to find any two different inputs x and y such that h(x) = h(y).
  + Since, hash function is compressing function with fixed hash length, it is impossible for a hash function not to have collisions. This property of collision free only confirms that these collisions should be hard to find.
  + This property makes it very difficult for an attacker to find two input values with the same hash.
  + Also, if a hash function is collision-resistant **then it is second pre-image resistant.**

More info: such as Applications of Hash Functions, Popular Hash functions, Desing of Hashing Algorithms.   
<https://www.tutorialspoint.com/cryptography/cryptography_hash_functions.htm>

### Binary search tree property

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

* The left subtree of a node contains only nodes with keys lesser 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.  
  There must be no duplicate nodes.



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

<https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/>

### Red-black-tree property

### Optimal Substructure Property

### Graph of dependencies

### Graphs, cycles on graphs, negative cycles, directed acyclic graphs

### Topological ordering on an acyclic graph

### shortest paths problems on graphs and weighted graphs

## Algorithms

### Insertion Sort

### Merge Sort

### Heapsort

### Quicksort

### Counting Sort

### Radix Sort

### Breadth-First Search

### Depth-first Search

### Topological Sorting

### Computing Strongly Connected Components

### Bellman-Ford Algorithm

### Dijkstra’s algorithm

### Floyd-Warshall algorithm

### Transitive closure of a directed graph

## Data Structures

### Max-Heaps & Min-Heaps

### Stacks

### Queues

### Linked Lists (both doubly-linked and singly-linked)

### Rooted Trees

### Hash Tables (direct-address, open addressing)

### Binary Search Trees

### Red-Black Trees

### Directed and Undirected Graphs (adjacency-list & adjacency-matrix representations)

### Weighted Graphs (adjacency-list & adjacency-matrix representations)

## Algorithm Design

### Divide and Conquer

### Dynamic Programming