CH 03 : REVIEW OF DATA STRUCTURES AND ALGORITHMIC TECHNIQUES

By Dr. LOUNNAS Bilal

CONTENTS

1 Introduction 3
2 Data structures 3
2.1 Data definition . . . 3
2.2 Basic operations . . . 5
2.3 Classification of data structure . . . 5
2.4 Pointer . . . 6
2.5 Array . . . 7
2.6 LinkedList . . . 10
2.7 Stack . . . 14
2.8 Queue . . . 18
2.9 Hash table . . . 22
3 Algorithmic techniques 26
3.1 Naïve algorithms . . . 26
3.2 Brute-Force algorithms . . . 28
3.3 Recursion . . . 30
3.4 Greedy Algorithms - "take what you can get now" strategy 32
3.5 Divide and conquer approach . . . 33
3.6 Conclusion . . . 34

LIST OF FIGURES

Figure 1 Example of Data type . . . 4
Figure 2 Type of data structure . . . 5
Figure 3 Pointer . . . 6
Figure 4 Pointer . . . 7
Figure 5 Array representation . . . 7
Figure 6 Array representation . . . 7
Figure 7 LinkedList representation . . . 10
Figure 8 Stack Example . . . 14
Figure 9 Stack representation . . . 15

Figure 10 Stack push operation 17
Figure 11 Stack pop operation 18
Figure 12 Queue Example 19
Figure 13 Queue representation 19
Figure 14 Enqueue operation 21
Figure 15 Dequeue operation 22
Figure 16 Hash function 23
Figure 17 Hash function with separate chaining 24
Figure 18 Hash function with open addressing 25
Figure 19 Comparison with Naïve algorithm and KFY algorithm for shuffling cards 28
Figure 20 brute force algorithm for string matching 29

LIST OF TABLES

Table 1 Characteristics of classification of data structure . 6

1 INTRODUCTION

In computer science, a data structure is a particular way of organizing data in a computer so that it can be used efficiently.

Data structures can implement one or more particular abstract data types (ADT), which specify the operations that can be performed on a data structure and the computational complexity of those operations. In comparison, a data structure is a concrete implementation of the specification provided by an ADT.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use B-tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers.

Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Data structures can be used to organize the storage and retrieval of information stored in both main memory and secondary memory.

The algorithmic techniques include: divide and conquer, backtracking, dynamic programming, greedy algorithms, hill-climbing ... ext.

Any solvable problem generally has at least one algorithm of each of the following types:

1. the obvious way.
2. the methodical way.
3. the clever way.
4. the miraculous way.

2 DATA STRUCTURES

2.1 Data definition

Data Definition defines a particular data with the following characteristics.

1. Atomic: Definition should define a single concept.
2. Traceable: Definition should be able to be mapped to some data element.
3. Accurate: Definition should be unambiguous.

4. Clear and Concise: Definition should be understandable.

Data Object represents an object having a data.
Data Type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types:

1. Built-in Data Type
2. Derived Data Type

2.1.1 *Built-in data type*

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

1. Integers
2. Boolean (true, false)
3. Floating (Decimal numbers)
4. Character and Strings

2.1.2 *Derived data type*

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example:

1. List
2. Array
3. Stack
4. Queue

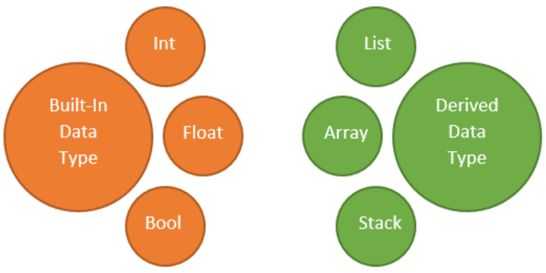


Figure 1: Example of Data type

2.2 Basic operations

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

1. Traversing
2. **Searching**
3. Insertion
4. Deletion
5. Sorting
6. Merging

2.3 Classification of data structure

Data structures are generally based on the effectiveness ability of a computer to fetch and store data at any place in its memory. Therefore, for an algorithm to be usable and efficient, it must retrieve data rapidlly and efficiently. The need for efficiency has led designers to use complex data structures to represent data.

There are numerous classification of data structures such as:

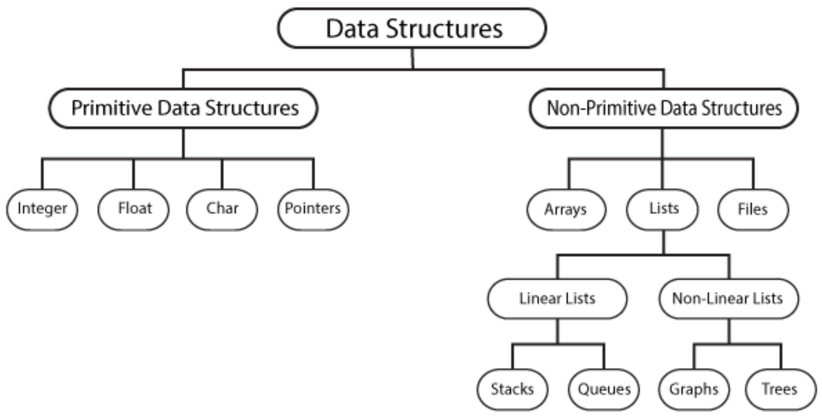


Figure 2: Type of data structure

The data structures can also be classified on the basis of the following characteristics: [See Table 1]

Characterstic	Description
Linear	In Linear data structures,the data items are arranged in a linear sequence. Example: Array
Non-Linear	In Non-Linear data structures,the data items are not in sequence. Example: Tree, Graph
Homogeneous	In homogeneous data structures,all the elements are of same type. Example: Array
Non-Homogeneous	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: Structures
Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: Array
Dynamic	Dynamic structures are those which expands or shrinks depending upon the,program need and its execution. Also, their associated memory locations,changes.,Example: Linked List created using pointers

Table 1: Characteristics of classification of data structure

2.4 Pointer

In computer science, a pointer is an object, whose value refers to (or "points to") another value stored elsewhere in the computer memory using its memory address. In other means it's a variable which contains the address in memory of another variable. We can have a pointer to any variable type.

Example of pointer that point to a normal variable [See Figure 3]

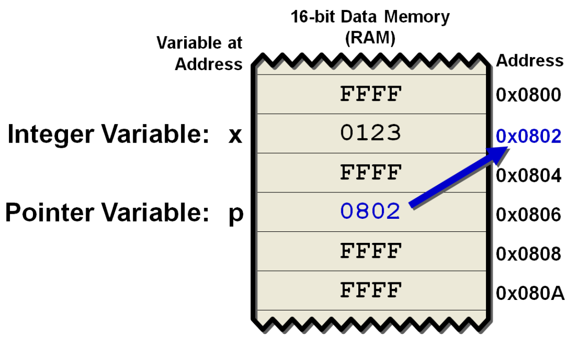


Figure 3: Pointer

When you use pointers, you are able to access the variable that they point to, and you are able to change the pointer so that it points to a different variable. This simple mechanism has hundreds of uses and is one of the most powerful features of the C programming language.

A pointer allows us to indirectly access a variable.

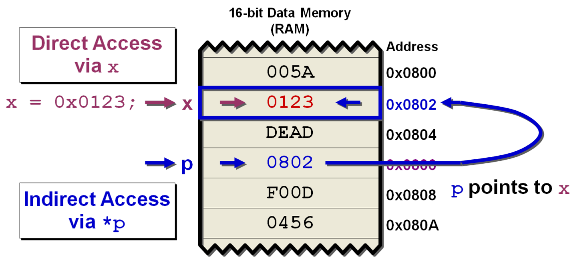


Figure 4: Pointer

2.5 Array

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

1. Element: Each item stored in an array is called an element.
2. Index: Each location of an element in an array has a numerical index, which is used to identify the element.

2.5.1 *Array representation*

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

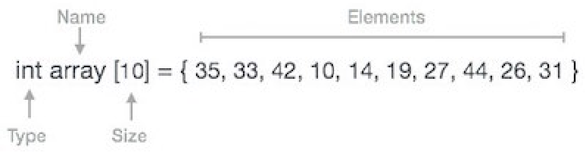


Figure 5: Array representation

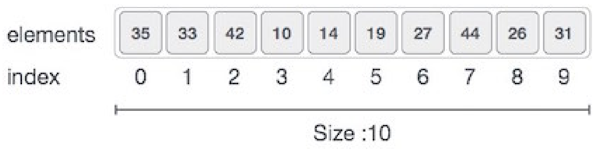


Figure 6: Array representation

As per the above illustration, following are the important points to be considered.

1. Index starts with 0.
2. Array length is 10 which means it can store 10 elements.
3. Each element can be accessed via its index. For example, we can fetch an element at index 6 or 9.

2.5.2 *Basic operations*

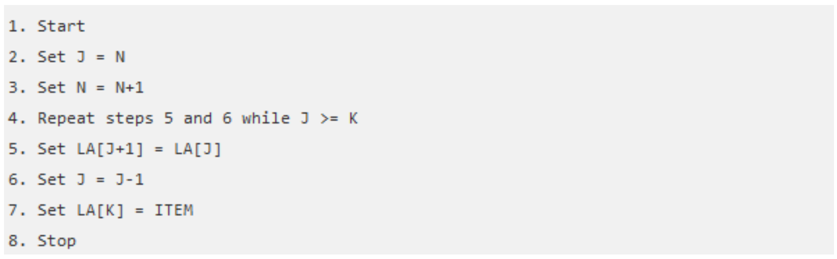
Following are the basic operations supported by an array.

1. Traverse: print all the array elements one by one.
2. Insertion: Adds an element at the given index.
3. Deletion: Deletes an element at the given index.
4. Search: Searches an element using the given index or by the value.
5. Update: Updates an element at the given index.

2.5.3 *Insertion operation*

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Algorithm
Let Array be a linear unordered array of MAX elements.
Let LA be a Linear Array (unordered) with N elements and K is a positive integer such that K<=N. Following is the algorithm where ITEM is inserted into the K^{th} position of LA.



```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

2.5.4 *Deletion operation*

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm
Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

2.5.5 *Search operation*

You can perform a search for an array element based on its value or its index.

Algorithm
Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

2.5.6 *Update operation*

Update operation refers to updating an existing element from the array at a given index..

Algorithm
Consider LA is a linear array with N elements and K is a positive integer such that K<=N. Following is the algorithm to update an element available at the Kth position of LA.

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

2.6 LinkedList

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- Link: Each link of a linked list can store a data called an element.
- Next: Each link of a linked list contains a link to the next link called Next.
- LinkedList: A Linked List contains the connection link to the first link called First.

2.6.1 *LinkedList representation*

Linked list can be visualized as a chain of nodes, where every node points to the next node.

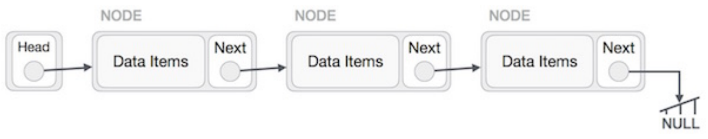


Figure 7: LinkedList representation

As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

2.6.2 *Types of LinkedList*

Following are the various types of linked list.

1. Simple Linked List: Item navigation is forward only.
2. Doubly Linked List: Items can be navigated forward and backward.
3. Circular Linked List: Last item contains link of the first element as next and the first element has a link to the last element as previous.

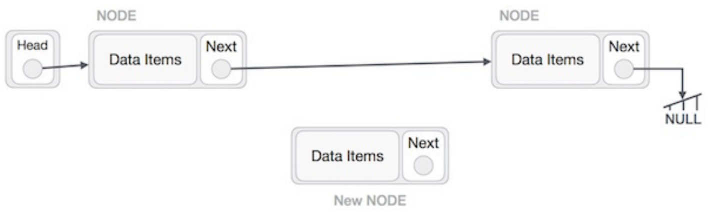
2.6.3 *Basic operations*

Following are the basic operations supported by a list.

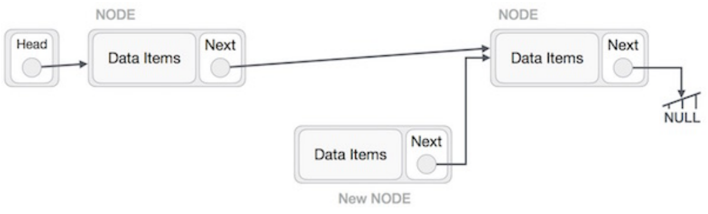
1. Insertion: Adds an element at the beginning of the list.
2. Deletion: Deletes an element at the beginning of the list.
3. Display: Displays the complete list.
4. Search: Searches an element using the given key.
5. Reverse: Reverse the whole linked list by making the last node pointing to the head node

2.6.4 *Insertion operation*

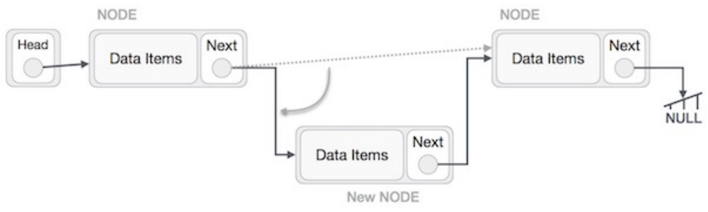
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



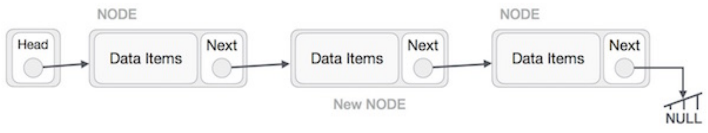
Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C. It should look like this:



Now, the next node at the left should point to the new node.



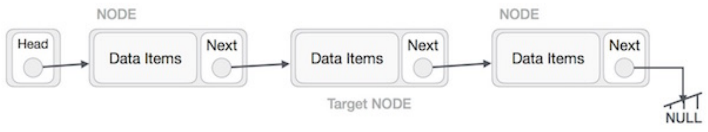
This will put the new node in the middle of the two. The new list should look like this:



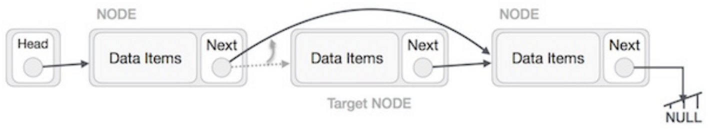
Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

2.6.5 *Deletion operation*

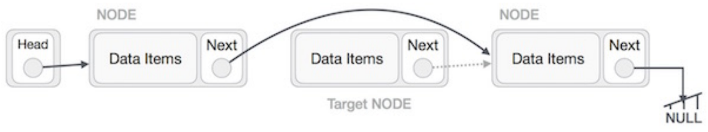
Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



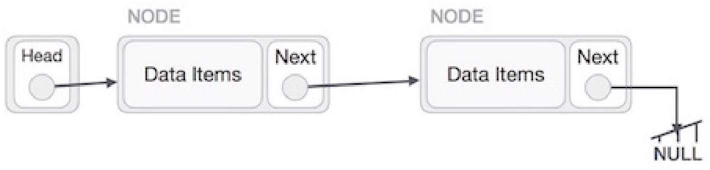
The left (previous) node of the target node now should point to the next node of the target node:



This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.



We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

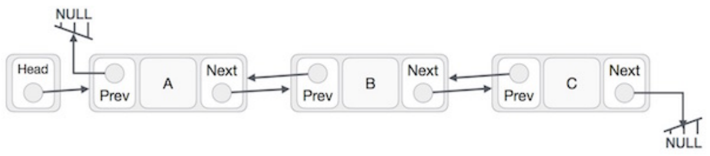


2.6.6 *Doubly LinkedList*

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

1. Link: Each link of a linked list can store a data called an element.
2. Next: Each link of a linked list contains a link to the next link called Next.
3. Prev: Each link of a linked list contains a link to the previous link called Prev.
4. LinkedList: A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly LinkedList representation



As per the above illustration, following are the important points to be considered.

1. Doubly Linked List contains a link element called first and last.
2. Each link carries a data field(s) and two link fields called next and prev.
3. Each link is linked with its next link using its next link.
4. Each link is linked with its previous link using its previous link.
5. The last link carries a link as null to mark the end of the list.

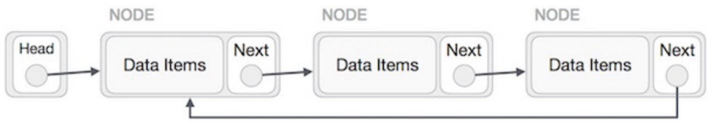
Basic operations
Following are the basic operations supported by a list.

1. Insertion: Adds an element at the beginning of the list.
2. Deletion: Deletes an element at the beginning of the list.
3. Insert Last: Adds an element at the end of the list.
4. Delete Last: Deletes an element from the end of the list.
5. Insert After: Adds an element after an item of the list.
6. Delete: Deletes an element from the list using the key.
7. Display forward: Displays the complete list in a forward manner.
8. Display backward: Displays the complete list in a backward manner.

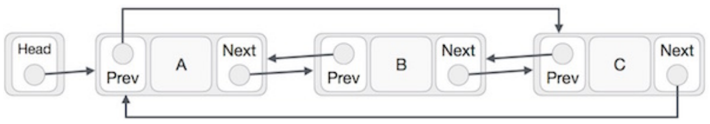
2.6.7 *Circular LinkedList*

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

Singly LinkedList as Circular
In singly linked list, the next pointer of the last node points to the first node.



Doubly LinkedList as Circular
In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



As per the above illustration, following are the important points to be considered.

1. The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
2. The first link's previous points to the last of the list in case of doubly linked list.

2.7 Stack

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example: a deck of cards or a pile of plates, etc.

Figure 8: Stack Example

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

2.7.1 *Stack representation*

The following diagram depicts a stack and its operations:

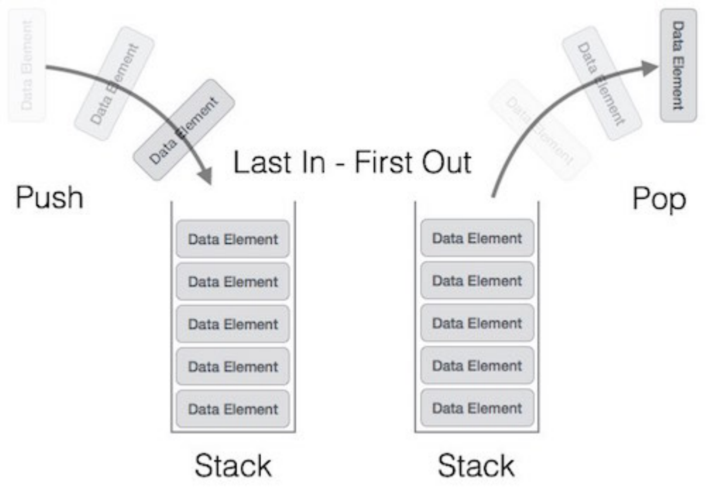


Figure 9: Stack representation

A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

2.7.2 *Basic operations*

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations:

1. push(): Pushing (storing) an element on the stack.
2. pop(): Removing (accessing) an element from the stack.

When data is PUSHed onto stack.
To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks:

1. peek(): get the top data element of the stack, without removing it.
2. isFull(): check if stack is full.
3. isEmpty(): check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions:

Peek()

```
begin procedure peek

   return stack[top]

end procedure
```

isfull()

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

isempty()

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

Push operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps:

1. Step 1: Checks if the stack is full.
2. Step 2: If the stack is full, produces an error and exit.
3. Step 3: If the stack is not full, increments top to point next empty space.
4. Step 4: Adds data element to the stack location, where top is pointing.
5. Step 5: Returns success.

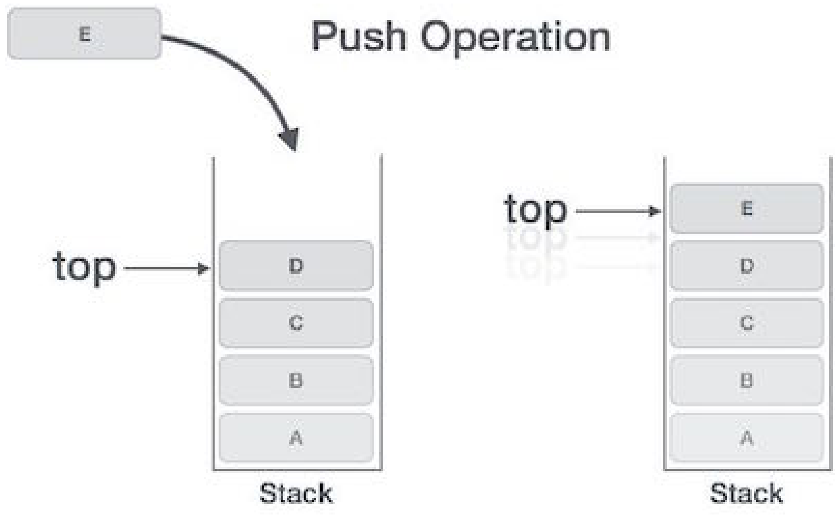


Figure 10: Stack push operation

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH operation
A simple algorithm for Push operation can be derived as follows

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1

   stack[top] ← data

end procedure
```

Pop operation
Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.
A Pop operation may involve the following steps:

1. Step 1: Checks if the stack is empty.
2. Step 2: If the stack is empty, produces an error and exit.
3. Step 3: If the stack is not empty, accesses the data element at which top is pointing.
4. Step 4: Decreases the value of top by 1.

5. Step 5: Returns success.

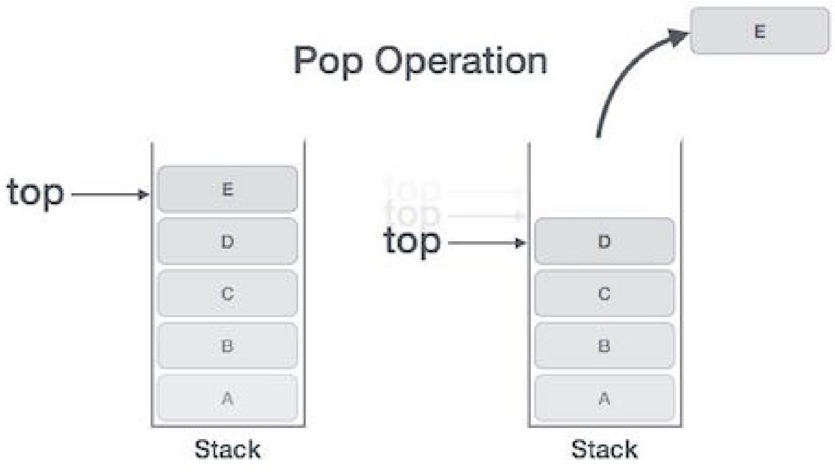


Figure 11: Stack pop operation

Algorithm for Pop operation
A simple algorithm for Pop operation can be derived as follows:

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]

   top ← top - 1

   return data

end procedure
```

2.8 Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

Figure 12: Queue Example

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

2.8.1 *Queue representation*

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure ?

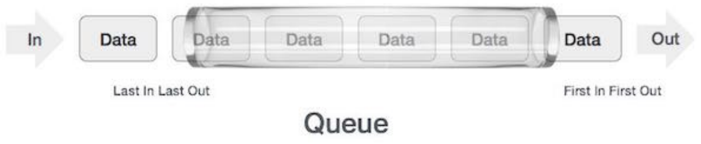


Figure 13: Queue representation

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

2.8.2 *Basic operations*

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues:

1. enqueue(): add (store) an item to the queue.
2. dequeue(): remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are:

1. peek(): Gets the element at the front of the queue without removing it.
2. isfull(): Checks if the queue is full.
3. isempty(): Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

Let's first learn about supportive functions of a queue:

Peek()
This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows:

```
begin procedure peek

   return queue[front]

end procedure
```

isfull()
As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ.

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

isempty()

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Enqueue operation
Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue:

1. Step 1: Check if the queue is full.
2. Step 2: If the queue is full, produce overflow error and exit.
3. Step 3: If the queue is not full, increment rear pointer to point the next empty space.
4. Step 4: Add data element to the queue location, where the rear is pointing.
5. Step 5: return success.

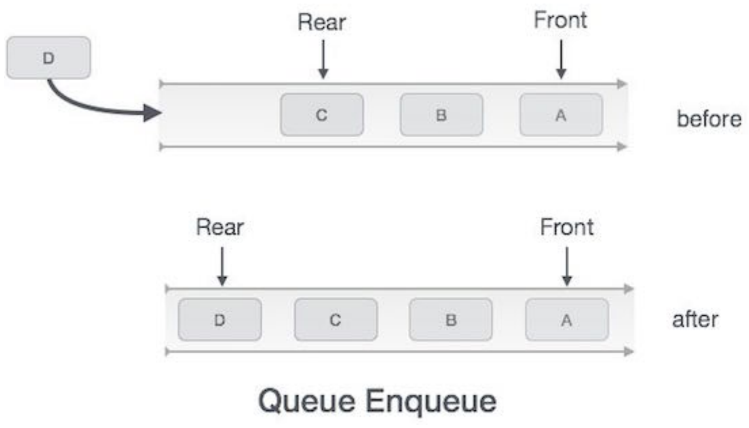


Figure 14: Enqueue operation

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

```
procedure enqueue(data)
   if queue is full
      return overflow
   endif

   rear ← rear + 1

   queue[rear] ← data

   return true

end procedure
```

Dequeue operation
Accessing data from the queue is a process of two tasks ? access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation:

1. Step 1: Check if the queue is empty.
2. Step 2: If the queue is empty, produce underflow error and exit.
3. Step 3: If the queue is not empty, access the data where front is pointing.
4. Step 4: Increment front pointer to point to the next available data element.
5. Step 5: Return success.

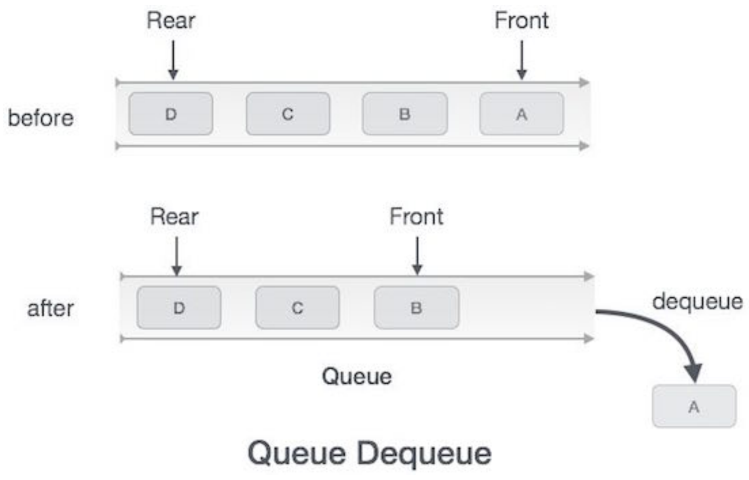


Figure 15: Dequeue operation

Dequeue algorithm

```
procedure dequeue
   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1

   return true
end procedure
```

2.9 Hash table

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

2.9.1 *Hashing*

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.

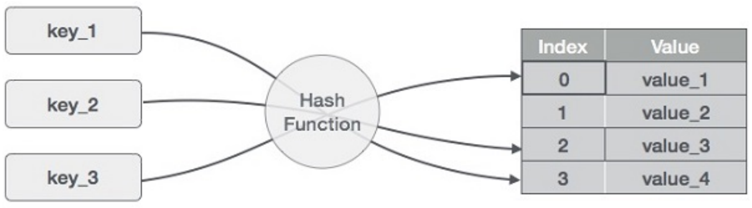


Figure 16: Hash function

Example:

Key	Hash	Array Index
1	1 % 20 = 1	1
2	2 % 20 = 2	2
42	42 % 20 = 2	2
4	4 % 20 = 4	4
12	12 % 20 = 12	12
14	14 % 20 = 14	14
17	17 % 20 = 17	17
13	13 % 20 = 13	13
37	37 % 20 = 17	17

As we can see, it may happen that the hashing technique is used to create an already used index of the array. This called collision.

Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,450 keys are hashed into a million buckets, even with a perfectly uniform random distribution, 95% chance of at least two of the keys being hashed to the same slot.

Therefore, almost all hash table implementations have some collision resolution strategy to handle such events.

Separate chaining (Open hashing)
In the method known as separate chaining, each bucket is independent, and has some sort of list of entries with the same index. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the list operation.

In a good hash table, each bucket has zero or one entries, and sometimes two or three, but rarely more than that. Therefore, structures that are efficient in time and space for these cases are preferred. Structures that are efficient for a fairly large number of entries per bucket are not needed or desirable. If these cases happen often, the hashing function needs to be fixed.

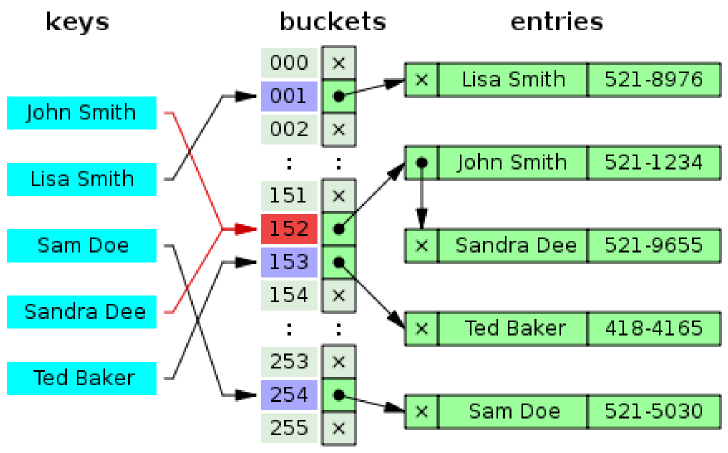


Figure 17: Hash function with separate chaining

Open addressing (Close hashing)
In another strategy, called open addressing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some probe sequence, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.

Well-known probe sequences include:

1. Linear probing, in which the interval between probes is fixed (usually 1).
2. Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation.
3. Double hashing, in which the interval between probes is computed by a second hash function.

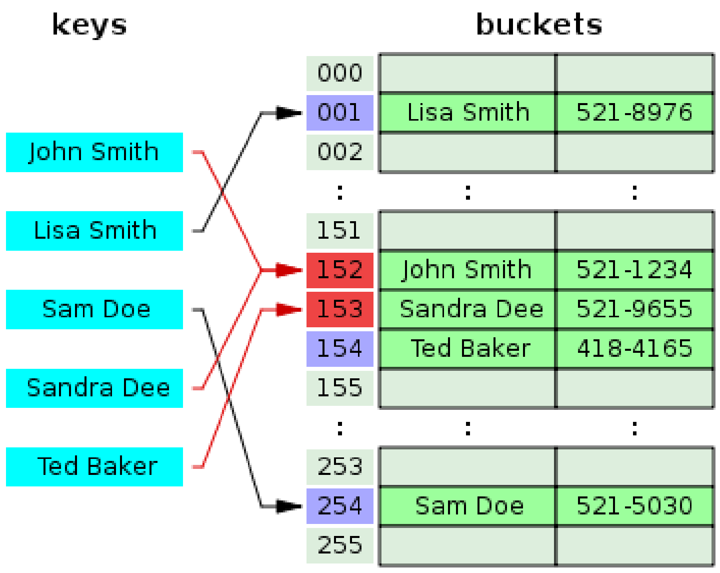


Figure 18: Hash function with open addressing

2.9.2 *Basic operations*

Following are the basic primary operations of a hash table.

1. Search: Searches an element in a hash table.
2. Insert: inserts an element in a hash table.
3. delete: Deletes an element from a hash table.

Search operation
Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

Insert operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Delete operation
Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

3 ALGORITHMIC TECHNIQUES

Any solvable problem generally has at least one algorithm of each of the following types:

1. the obvious way.
2. the methodical way.
3. the clever way.
4. the miraculous way.

On the first and most basic level, the "obvious" solution might try to exhaustively search for the answer. Intuitively, the obvious solution is the one that comes easily if you're familiar with a programming language and the basic problem solving techniques.

The second level is the methodical level and is the heart of this book: after understanding the material presented here you should be able to methodically turn most obvious algorithms into better performing algorithms.

The third level, the clever level, requires more understanding of the elements involved in the problem and their properties or even a reformulation of the algorithm (e.g., numerical algorithms exploit mathematical properties that are not obvious). A clever algorithm may be hard to understand by being non-obvious that it is correct, or it may be hard to understand that it actually runs faster than what it would seem to require.

The fourth and final level of an algorithmic solution is the miraculous level: this is reserved for the rare cases where a breakthrough results in a highly non-intuitive solution.

3.1 Naïve algorithms

A naïve algorithm is a very simple solution to a problem. It is meant to describe a suboptimal algorithm compared to a "clever" (but less simple) algorithm. Naïve algorithms usually consume larger amounts of resources (time, space, memory accesses, ...), but are simple to devise and implement.

3.1.1 *Examples of Naïve algorithm*

Here's an examples of some problems with a "naïve" algorithms:

1. An example of a na algorithm is bubble sort, which is only a few lines long and easy to understand, but has a $O(n^2)$ time complexity.
2. Trying to search for an element in a sorted array. A Naive algorithm would be to use a Linear Search. A Not-So Naive Solution would be to use the Binary Search.
3. **Problem: You are in a (2-dimensional) maze. Find your way out. (meaning: to a spot with an "EXIT" sign**

   Naïve algorithm 1: Start walking and choose the right one in every intersection you meet (until you find "EXIT").

   Naïve algorithm 2:: Start walking and choose a random one in every intersection you meet (until you find "EXIT").

   Algorithm 1 will not even get you out of some mazes!

   Algorithm 2 will get you out of all mazes (although this is rather hard to prove).

4. For instance, if one knows the definition of Fibonacci numbers is **Fib(n)=Fib(n-1)+Fib(n-2)**, then a "naïve" implementation would be

```
def Fib(n):
  if n <= 1:
    return 1
  else:
    return Fib(n-1) + Fib(n-2)
```

3.1.2 *Issues with a Naïve algorithms*

Sometimes there are some problems with the naïve algorithms, a problems that most programmers won't see, for example if we take example (4) above:

What if we call, say, Fib(7), then we end up making many of the same calls over and over, such as Fib(4) (because Fib(7) calls Fib(6) and Fib(5), and Fib(6) calls Fib(5) and Fib(4), and both times we call Fib(5) it calls Fib(4) and Fib(3), and so on).

Usually, naïve algorithms are not wrong, just oversimplified and inefficient. The danger, in this case, is rather they give a stabilized solution naïve algorithms sometime give an unexpected results.

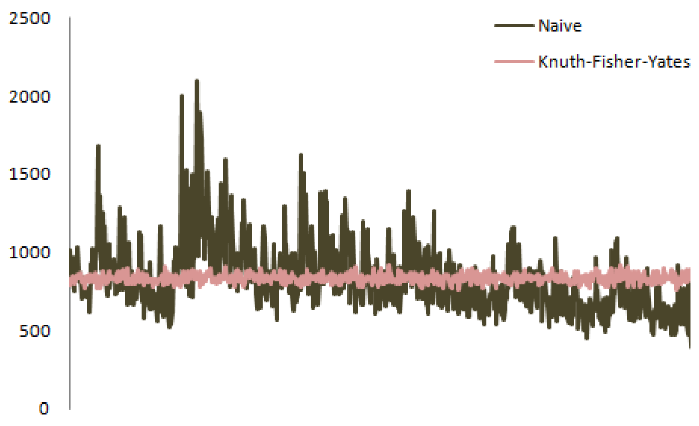


Figure 19: Comparison with Naïve algorithm and KFY algorithm for shuffling cards

Naïve implementations are often preferred to complex ones. Simplicity is a virtue. It's better to be simple, slow, and understandable than complex, fast, and difficult to grasp. Or at least it usually is. Sometimes, the simplicity of the na implementation can mislead. It is possible for the code to be both simple and wrong. We suppose the real lesson lies in testing. No matter how simple your code may be, there's no substitute for testing it to make sure it's actually doing what you think it is.

3.2 Brute-Force algorithms

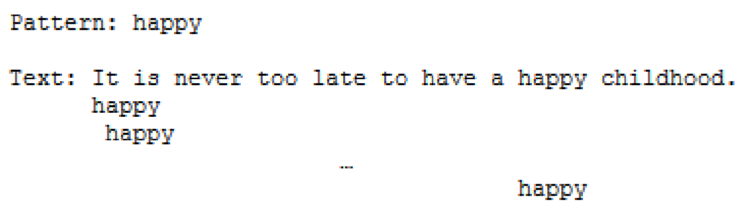
Brute force is a straightforward approach to solve a problem based on the problems statement and definitions of the concepts involved. It is considered as one of the easiest approach to apply and is useful for solving smallsize instances of a problem.

While a brute-force is simple to implement, and will always find a solution if it exists, its cost is proportional to the number of candidate solutions - which in many practical problems tends to grow very quickly as the size of the problem increases. Therefore, brute-force is typically used when the problem size is limited, or when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size. The method is also used when the simplicity of implementation is more important than speed.

3.2.1 *Brute-Force String Matching*

A brute force algorithm for string matching problem has two inputs to be considered: pattern (a string of m characters to search for), and text (a long string of n characters to search in). The algorithm starts with aligning the pattern at the beginning of the text. Then each character of the pattern is compared to the corresponding character, moving from left to right, until all characters are found to match, or a mismatch is detected.

While the pattern is not found and the text is not yet exhausted, the pattern is realigned to one position to the right and again compared to the corresponding character, moving from left to right.



```
Pattern: happy

Text: It is never too late to have a happy childhood.
      happy
       happy
                    ...
                                    happy
```

Figure 20: brute force algorithm for string matching

Some examples of brute force algorithms are:

1. Computing an (a > 0, n a nonnegative integer) by multiplying a*a**a
2. Computing n!
3. Selection sort
4. Bubble sort
5. Sequential search
6. Exhaustive search: Traveling Salesman Problem, Knapsack problem.

3.2.2 *Advantages and disadvantages*

The strengths of using a brute force approach are as follows:

1. It has wide applicability and is known for its simplicity.
2. It yields reasonable algorithms for some important problems such as searching, string matching, and matrix multiplication.
3. It yields standard algorithms for simple computational tasks such as sum and product of n numbers, and finding maximum or minimum in a list.

The weaknesses of the brute force approach are as follows:

1. It rarely yields efficient algorithms.

2. Some brute force algorithms are unacceptably slow.
3. It is neither as constructive nor creative as some other design techniques.

3.3 Recursion

Have you ever seen a set of Russian dolls? At first, you see just one figurine, usually painted wood, that looks something like this:

You can remove the top half of the first doll, and what do you see inside? Another, slightly smaller, Russian doll!, You can remove that doll and separate its top and bottom halves. And you see yet another, even smaller, doll. And once more, And another once more. And you can keep going. Eventually you find the teeniest Russian doll. It is just one piece, and so it does not open:

What do Russian dolls have to do with algorithms? Just as one Russian doll has within it a smaller Russian doll, which has an even smaller Russian doll within it, all the way down to a tiny Russian doll that is too small to contain another, we'll see how to design an algorithm to solve a problem by solving a smaller instance of the same problem, unless the problem is so small that we can just solve it directly. We call this technique recursion.

A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem.

In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem.

Example 1: Algorithm for finding the k-th even natural number

Note here that this can be solved very easily by simply outputting 2*(k - 1) for a given k . The purpose here, however, is to illustrate the basic idea of recursion rather than solving the problem.

Algorithm 1: Algorithm for finding the k-th even natural number

```
Even (positive integer k);
Input   : k , a positive integer
Output: k-th even natural number (the first even being 0)
if k = 1 then
    return 0;
else
    return Even(k-1) + 2;
end
```

Example 2: Algorithm for computing the k-th power of 2

Algorithm 2: Algorithm for computing the k-th power of 2

```
Powerof2 (natural number k);
Input   : k , a natural number
Output: k-th power of 2
if k = 0 then
    return 1;
else
    return 2*Powerof2(k - 1);
end
```

3.3.1 *Types of Recursion*

1. Linear recursion : makes at most one recursive call each time it is invoked.
2. Binary recursion : algorithm makes two recursive calls.
3. Multiple recursion : method may make (potentially more than two) recursive calls.

3.3.2 *Properties of Recursion*

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have:

1. Base criteria: There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
2. Progressive approach: The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

3.4 Greedy Algorithms - "take what you can get now" strategy

An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

3.4.1 *Counting Coins*

This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of € 1, 2, 5 and 10 and we are asked to count €18 then the greedy procedure will be:

1. 1: Select one € 10 coin, the remaining count is 8
2. 2: Then select one € 5 coin, the remaining count is 3
3. 3: Then select one € 2 coin, the remaining count is 1
4. 3: And finally, the selection of one € 1 coins solves the problem.

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1) Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

3.4.2 *Examples*

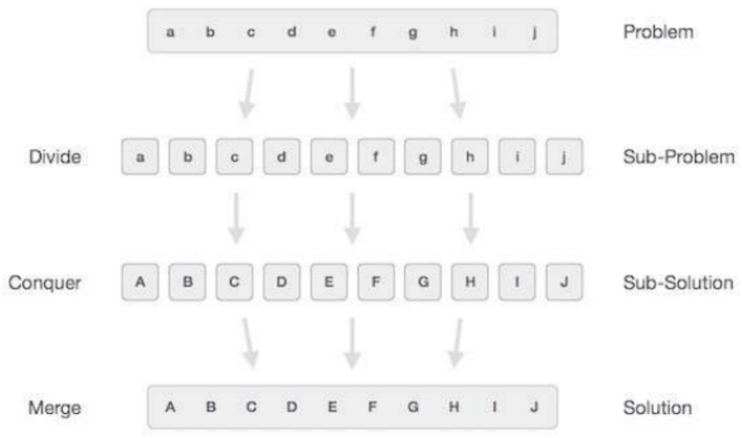
Most networking algorithms use the greedy approach. Here is a list of few of them:

1. Travelling Salesman Problem
2. Prim's Minimal Spanning Tree Algorithm
3. Kruskal's Minimal Spanning Tree Algorithm
4. Dijkstra's Minimal Spanning Tree Algorithm
5. Graph: Map Coloring
6. Graph: Vertex Cover
7. Knapsack Problem
8. Job Scheduling Problem

There are lots of similar problems that uses the greedy approach to find an optimum solution.

3.5 Divide and conquer approach

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.



3.5.1 *Divide/Break*

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

3.5.2 *Conquer/Solve*

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

3.5.3 *Merge/Combine*

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer merge steps works so close that they appear as one.

3.5.4 *Examples*

The following computer algorithms are based on divide-and-conquer programming approach:

1. Merge Sort
2. Quick Sort
3. Binary Search
4. Strassen's Matrix Multiplication
5. Closest Pair (points)

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

3.6 Conclusion

Selecting a proper designing technique for a algorithm is the most difficult and important task. Most of the programming problems may have more than one solution, for that we presented some algorithm techniques that are highly used in creating solution.

the following designing techniques are not mentioned in this textbook

1. Dynamic Programming
2. Backtracking Algorithm
3. Branch and Bound
4. Linear Programming
5. Transform-and-Conquer
6. Decrease-and-Conquer
7. ...