Week 8 Lecture 1

Class	BSCCS2001
Created	@October 24, 2021 6:21 PM
Materials	
Module #	36
Type	Lecture
Week #	8

Algorithms and Data Structures: Algorithms and Complexity Analysis

Algorithms and Programs

- **Algorithms**
  - An algorithm is a finite sequence of well-defined, computer-implementable (optional) instructions, typically solves a class of specific problems or to perform a computation
  - Algorithms are always un-ambiguous and are used as specifications for performing calculations, data processing, automated reasoning and other tasks
  - An algorithm must terminate
- **Program**
  - A computer program is a collection of instructions that can be executed by a computer to perform a specific task
  - A computer program is usually written by a computer programmer in a programming language
  - A program implements an algorithm
  - A program may or may not terminate
    - For example → An Operating System

Analysis of Algorithms

- **Why?**
  - Set the motivation for algorithm analysis

    - Why analyze?
- **What?**
    - Identify what all needs to be analyzed
    - What to analyze?
- **How?**
    - Learn the techniques for analysis
    - How to analyze?
- **Where?**
    - Understand the scenarios for application
    - Where to analyze?
- **When?**
    - Realize your position for seeking the analysis
    - When to analyze?

Why analyze?

Practical reasons:

- Resources are scarce
- Greed to do more with less
- Avoid performance bugs

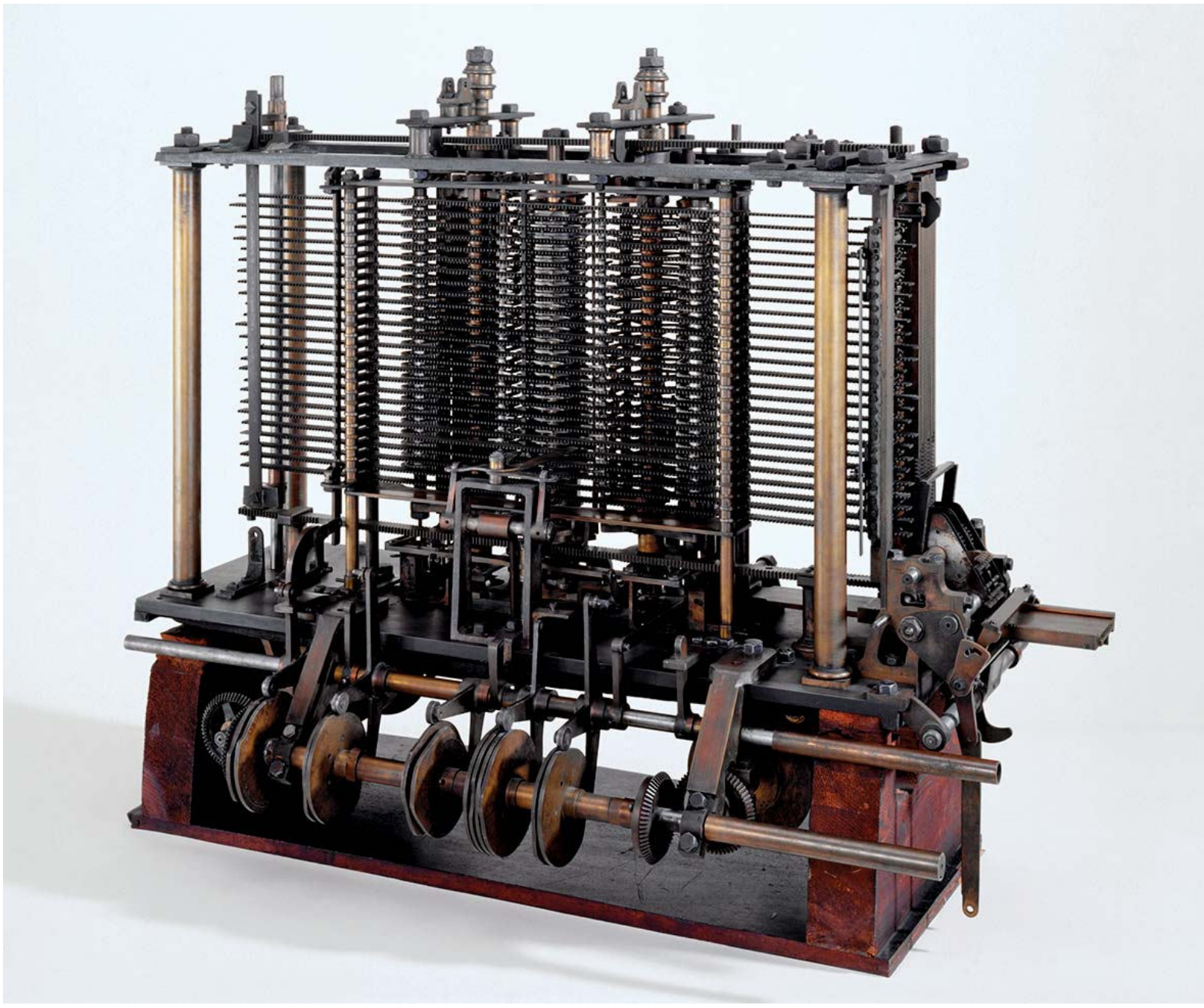
Core Issues:

- Predict performance
    - How much time does binary search take?
- Compare algorithms
    - How quick is Quicksort? (heh)
- Provide guarantees
    - Size notwithstanding, Red-Black tree inserts in $O(log\ n)$
- Understand theoretical basis
    - Sorting by comparison cannot do better than $\Omega(n\ log\ n)$

What to analyze?

Core Issues → Cannot control what we cannot measure

- Time
    - Story starts with the Analytical Engine



- Most common analysis factor
- Representative of various related analysis factors like Power, Bandwidth, Processors
- Supported by Complexity Classes

- Space
  - Widely exported
  - Important for hand-held devices
  - Supported by Complexity Classes

What to analyze?

- Sum of natural numbers

```
int sum(int n)
{
  int s = 0;
  for(; n > 0; --n)
    s = s + n;
  return s;
}
```

- Time $T(n) = n$ (additions)
- Space $S(n) = 2$ (n, s)

What to analyze?

- Find a character in a string

```
int find(char *str, char c)
{
  for(int i = 0; i < strlen(str); ++i)
    if(str[i] == c)
      return i;
```

```
  return 0;
}
n = strlen(str)
```

- Time $T(n) = n$ (compare) + $n * T(strlen(str)) \approx n + n^2 \approx n^2$
- Space $S(n) = 3$ (str, c, i)

What to analyze?

- Minimum of a sequence of numbers

```
int min(int a[], int n)
{
  for(int i = 0; i < n; ++i)
    cin >> a[i];

  int t = a[--n];
  for(; n > 0; --n)
    if(t < a[--n])
      t = a[n];
  return t;
}
```

- Time $T(n) = n - 1$ (comparison of value)
- Space $S(n) = n + 3$ (`a[]`'s, n, i, t)

How to analyze?

- Counting model
- Asymptotic model
- Generating functions
- Master Theorem

How to analyze? Counting Models

- **Core Idea** → Total running time = Sum of cost $\times$ frequency for all operations
  - Need to analyze program to determine set of operations
  - Cost depends on machine, compiler
  - Frequency depends on the algorithm, input data
- **Machine Model** → Random Access Machine (RAM) Computing Model
  - Input data & size
  - Operations
  - Intermediate Stages
  - Output data & size

How to analyze? Counting Models

- **Factorial (Recursive)**

```
int fact(int n)
{
  if (n != 0)
    return n * fact(n - 1);
  return 1;
}
```

  - Time $T(n) = n - 1$ (multiplication)
  - Space $S(n) = n + 1$ (n's in recursive calls)
- **Factorial (Iterative)**

```
int fact(int n)
{
  int t = 1;
  for(; n > 0; --n)
    t = t * n;
  return t
}
```

- Time $T(n) = n$ (multiplication)
- Space $S(n) = 2$ (`n, t`)

How to analyze? Asymptotic Analysis

Asymptotic Analysis

- **Core Idea** → Cannot compare actual times; hence, compare Growth or how the time increases with input size
  - Function Approximation (tilde $(\tilde{})$ notation)
  - Common growth functions
    - Big-Oh → $O(.)$
    - Big-Omega → $\Omega(.)$
    - Big-Theta $\Theta(.)$
  - Solve recurrence with Growth functions

How to analyze? Asymptotic Analysis

```
int count = 0;
for(int i = 0; i < N; ++i)
  for(int j = i + 1; i < N; ++j)
    if (a[i] + a[j] == 0)
      count++;
```

Function Approximation (tilde ($\tilde{}$) notation)

Operation	Frequency	Approximation
variable declaration	$N + 2$	$\sim N$
assignment statement	$N + 2$	$\sim N$
less than compare	$\frac{1}{2}(N + 1)(N + 2)$	$\sim \frac{1}{2}N^2$
equal to compare	$\frac{1}{2}N(N - 1)$	$\sim \frac{1}{2}N^2$
array access	$N(N - 1)$	$\sim N^2$
increment	$\frac{1}{2}N(N - 1)$ to $N(N - 1)$	$\sim \frac{1}{2}N^2$ to $\sim N^2$

- Estimate running time (or memory) as a function of input size N
- Ignore lower order terms
  - When N is large, terms are negligible
  - When N is small, we don't care

$f(n) \sim g(n)$ **means**

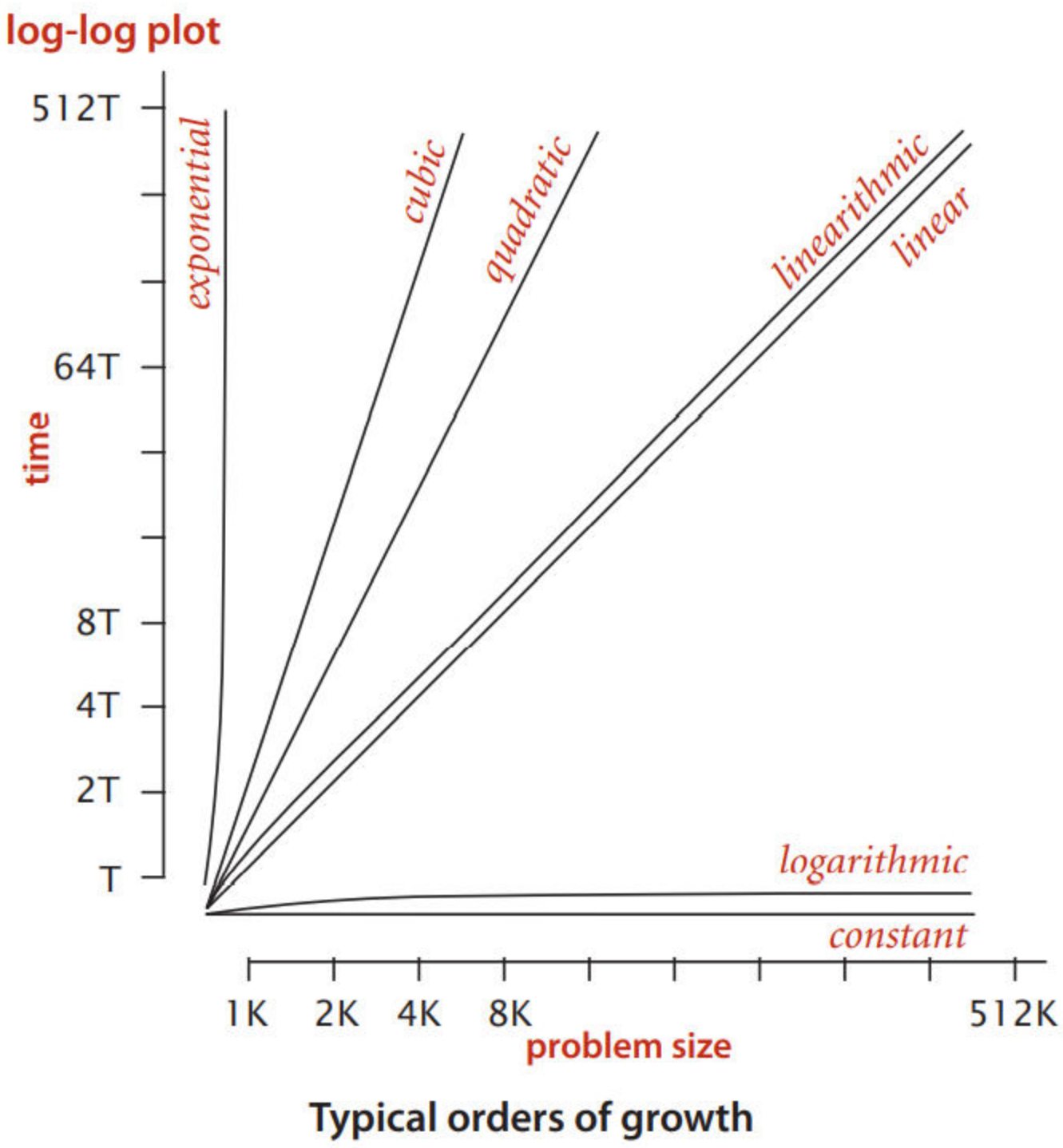
$\lim_{N \to \infty} \frac{f(n)}{g(n)} = 1$

How to analyze? Asymptotic Analysis

Common order-of-growth classifications

Good news. The set of functions

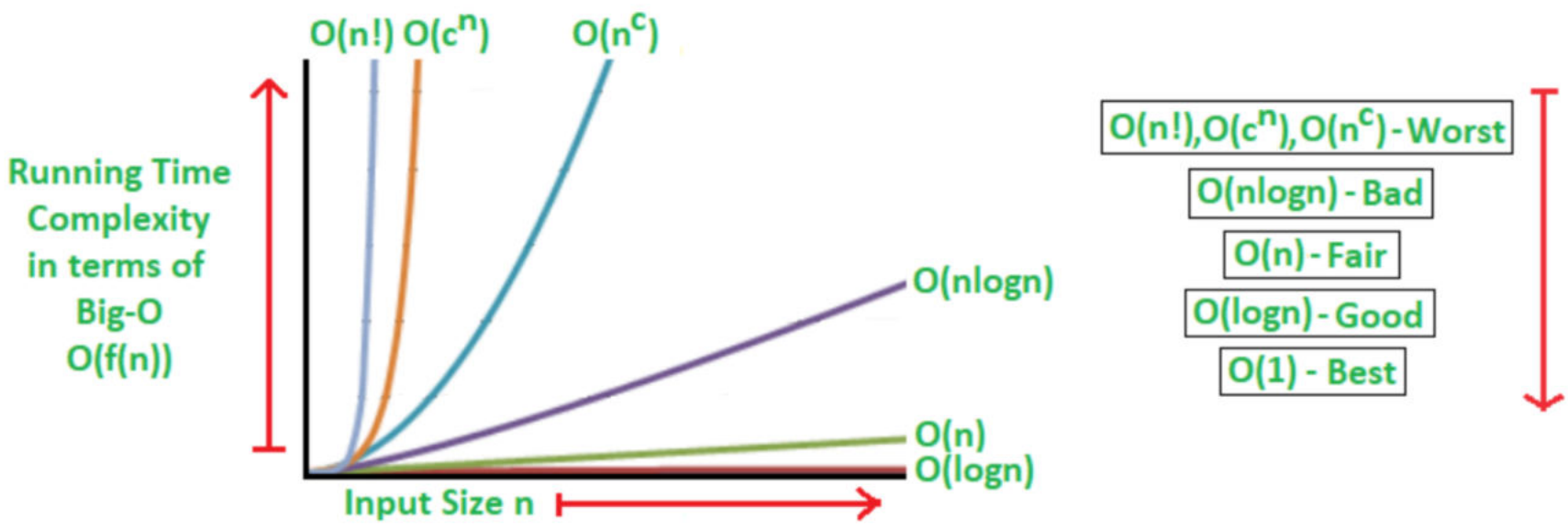
$1, logN, N, NlogN, N^2, N^3, and, 2^N$ suffices to describe to describe the order of growth of most common algorithms



Typical orders of growth

Source: Page #188, Chapter 1: Fundamentals, Section 1.4, Algorithms (4th Edition) by Robert Sedgewick & Kevin Wayne

How to analyze? Asymptotic Analysis



How to analyze? Asymptotic Analysis

description	order of growth	typical code framework	description	example
constant	1	`a = b + c;`	*statement*	*add two numbers*
logarithmic	$\log N$	[*see page 47*]	*divide in half*	*binary search*

linear	N	`double max = a[0];`
		`for (int i = 1; i < N; i++)`
		`  if (a[i] > max) max = a[i];`
			loop	*find the maximum*

linearithmic	$N \log N$	[*see* ALGORITHM 2.4]	*divide and conquer*	*mergesort*

quadratic	N^2	`for (int i = 0; i < N; i++)`
		`  for (int j = i+1; j < N; j++)`
		`  if (a[i] + a[j] == 0)`
		`  cnt++;`
			double loop	*check all pairs*

cubic	N^3	`for (int i = 0; i < N; i++)`
		`  for (int j = i+1; j < N; j++)`
		`  for (int k = j+1; k < N; k++)`
		`  if (a[i] + a[j] + a[k] == 0)`
		`  cnt++;`
			triple loop	*check all triples*

exponential	2^N	[*see* CHAPTER 6]	*exhasutive search*	*check all subsets*

Summary of common order-of-growth hypotheses

Source: Page #187, Chapter 1: Fundamentals, Section 1.4, Algorithms (4th Edition) by Robert Sedgewick & Kevin Wayne

Asymptotic Notation

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions:

$O(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$, for all $n > n_0\}$

- We use O-notation to give an upper bound on a function, to within a constant factor
- When we say that the running time of A is $O(n^2)$, we mean that there is a function $f(n)$ that is $O(n^2)$ such that for any value of n, no matter what particular input of size n is chosen, the running time on that input is bounded from above by the value $f(n)$
- Equivalently, we mean that the worst-case running time is $O(n^2)$

Where to analyze?

Algorithmic situation

- **Core Idea** → Identify data configurations or scenarios for analysis
  - Best case
    - Minimum running time on an input
  - Worst case
    - Running time guarantees for any input of size n
  - Average case
    - Expected running time for a random input of size n

- Probabilistic case
  - Expected running time of a randomized algorithm
- Amortized case
  - Worst case running time for any sequence of n operations

Big-O Algorithm Complexity Chart

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

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)

Source: https://www.bigocheatsheet.com

Week 8 Lecture 2

Class	BSCCS2001
Created	@October 25, 2021 12:10 AM
Materials	
Module #	37
Type	Lecture
Week #	8

Algorithms and Data Structures: Data Structures

Data Structure

- A data structure specifies the way of organizing and storing in-memory data that enables efficient access and modification of the data
    - Linear Data Structures
    - Non-linear Data Structures
- Most data structure has a container for the data and typical operations that it needs to perform
- For applications relating to data management, the key operations are:
    - Create
    - Insert
    - Delete
    - Find/Search
    - Close
- Efficiency is measured in terms of time and space taken for these operations

Linear Data Structures

- A linear data structure has data elements arranged in linear of sequential manner such that each member element is connected to its previous and next element
- Since data elements are sequentially connected, each element is traversable through a single run

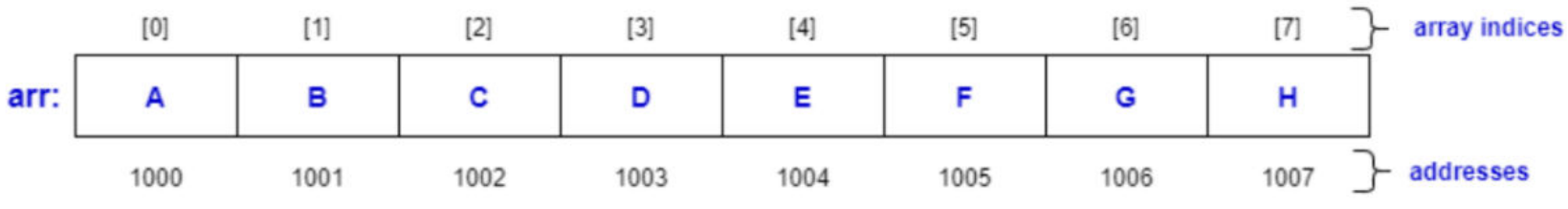
- Examples of linear data structures are Array, Linked List, Queue, Stack, etc

Different examples of linear data structures:

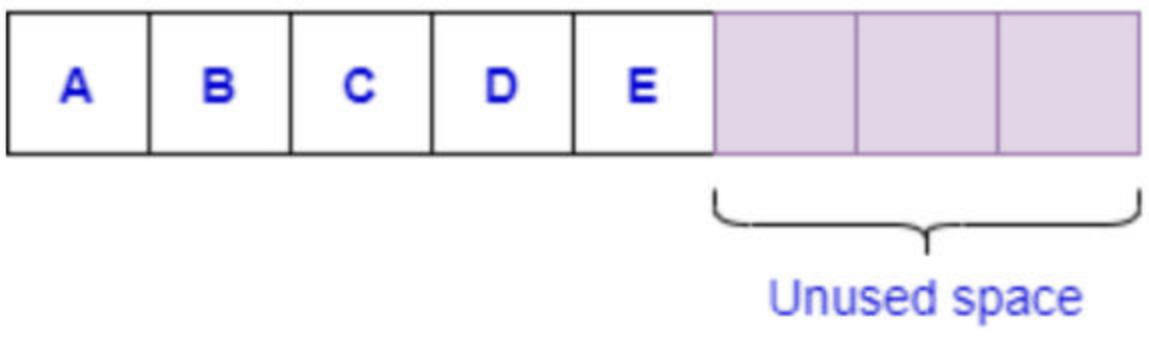
- **Array** → The data elements are stored at contiguous locations in the memory
- **Linked List** → The data elements are not required to be stored in contiguous locations in memory
  - Rather, each element stores a link (a pointer to a reference) to the location of the next element
- **Queue** → It is a FIFO (First In, First Out) data structure
  - The element that has been inserted first in the queue would be removed first
  - Thus, insert and removal of the elements in this take place in the same order
- **Stack** → It is a LIFO (Last In, First Out) data structure
  - The element that has been inserted last in the stack would be removed first
  - Thus, insert and removal of the elements in this take place in the reverse order

Linear Data Structure: Array

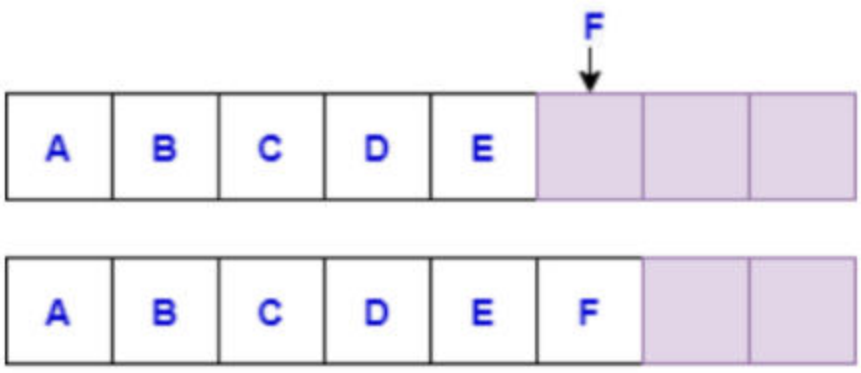
- The elements are stored in contiguous memory locations



- Simple access using indices
  - For example → let the array name be `arr`, we can access the element at index 5 as `arr[5]`
- Arrays allow random access using its index which is fast (cost of $O(1)$)
  - Useful for operations like sorting, searching
- **<u>Have fixed size, not flexible →</u>** Since we do not know the number of elements to be stored in runtime, If we create it too large then it can be a waste of memory, if we create it too small then some elements may not be accommodated in the array
  - *For example* → Suppose we create an array to store 8 elements
    - However, during the execution of the program only 5 elements are available, which results in the wastage of the memory space

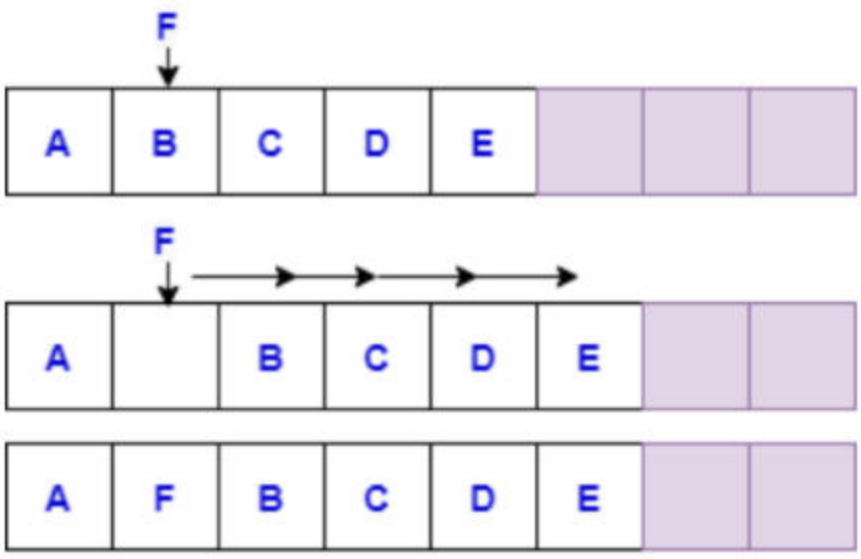


- Insertion and removal of elements from an array are costlier since the memory locations have to be consecutive
  - Insertion or removal of an element from the end of an array is easy
    - Insert at the end:

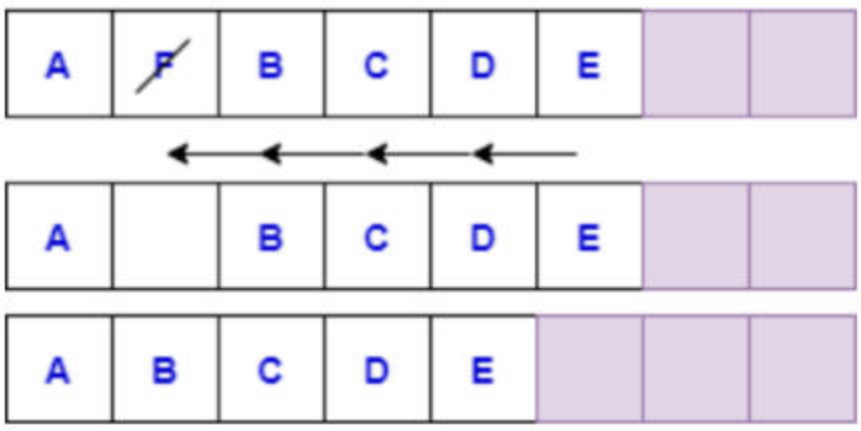


- Remove from the end:

- Insert and remove elements at any arbitrary position is costly (cost of $O(n)$)
- Insert at any arbitrary position

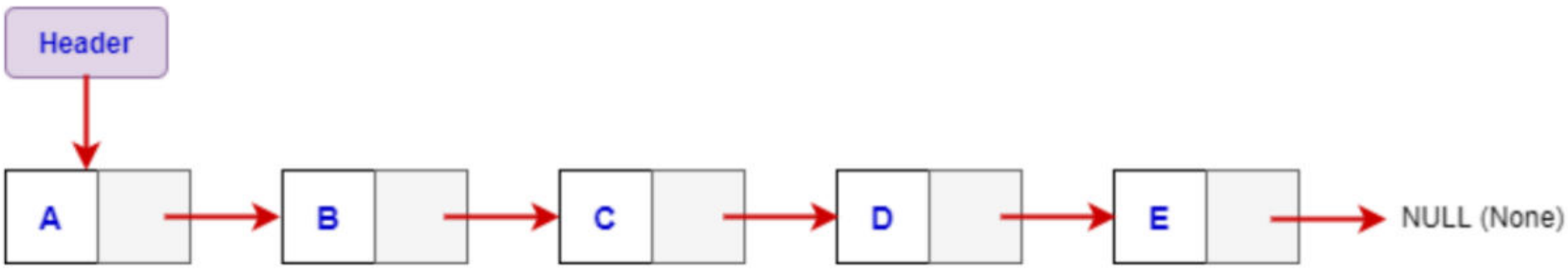


- Remove from any arbitrary position



Linear Data Structure: Linked List

- Elements are not required to be stored at contiguous memory locations
  - A new element can be stored anywhere in the memory where free space is available
  - Thus, it provides better memory usage than arrays
- For each new element allocated, a link (a pointer or a reference) is created for the new element using which the element can be added to the linked list



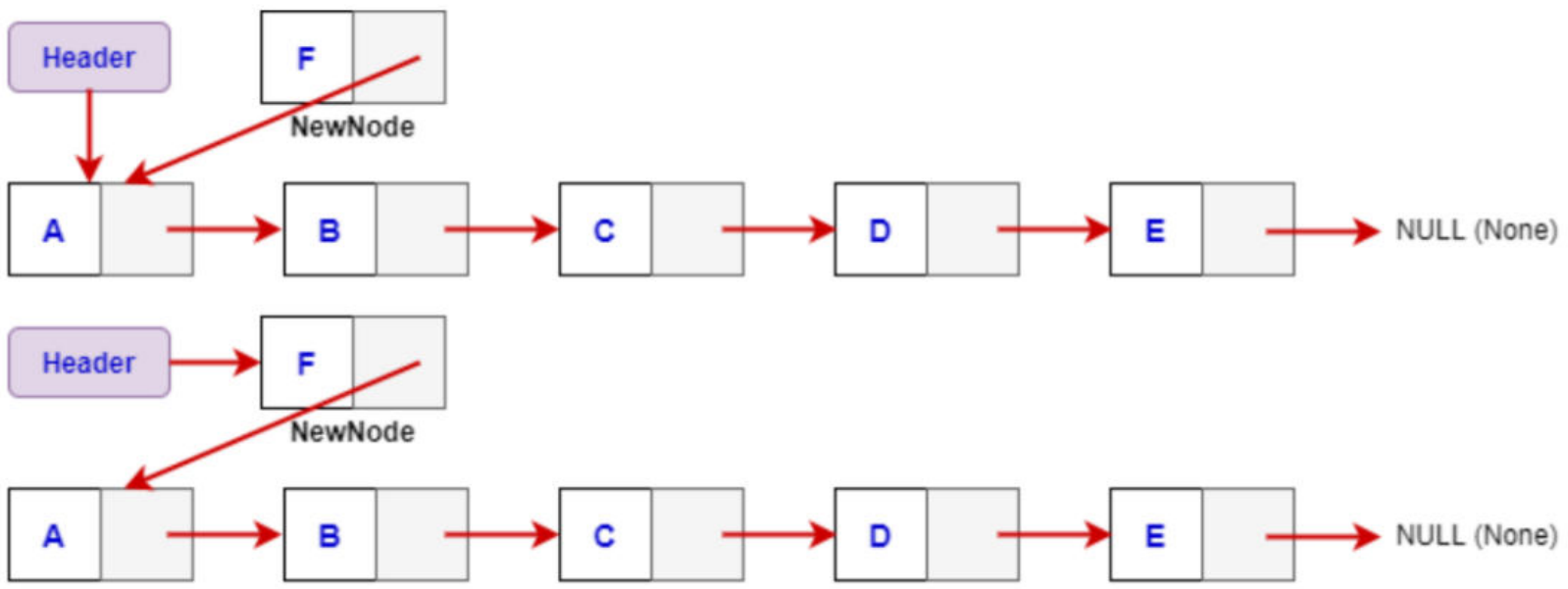
Each element is stored in a node

A node has 2 parts

- **Info** → stores the element
- **Link** → stores the location of the next node

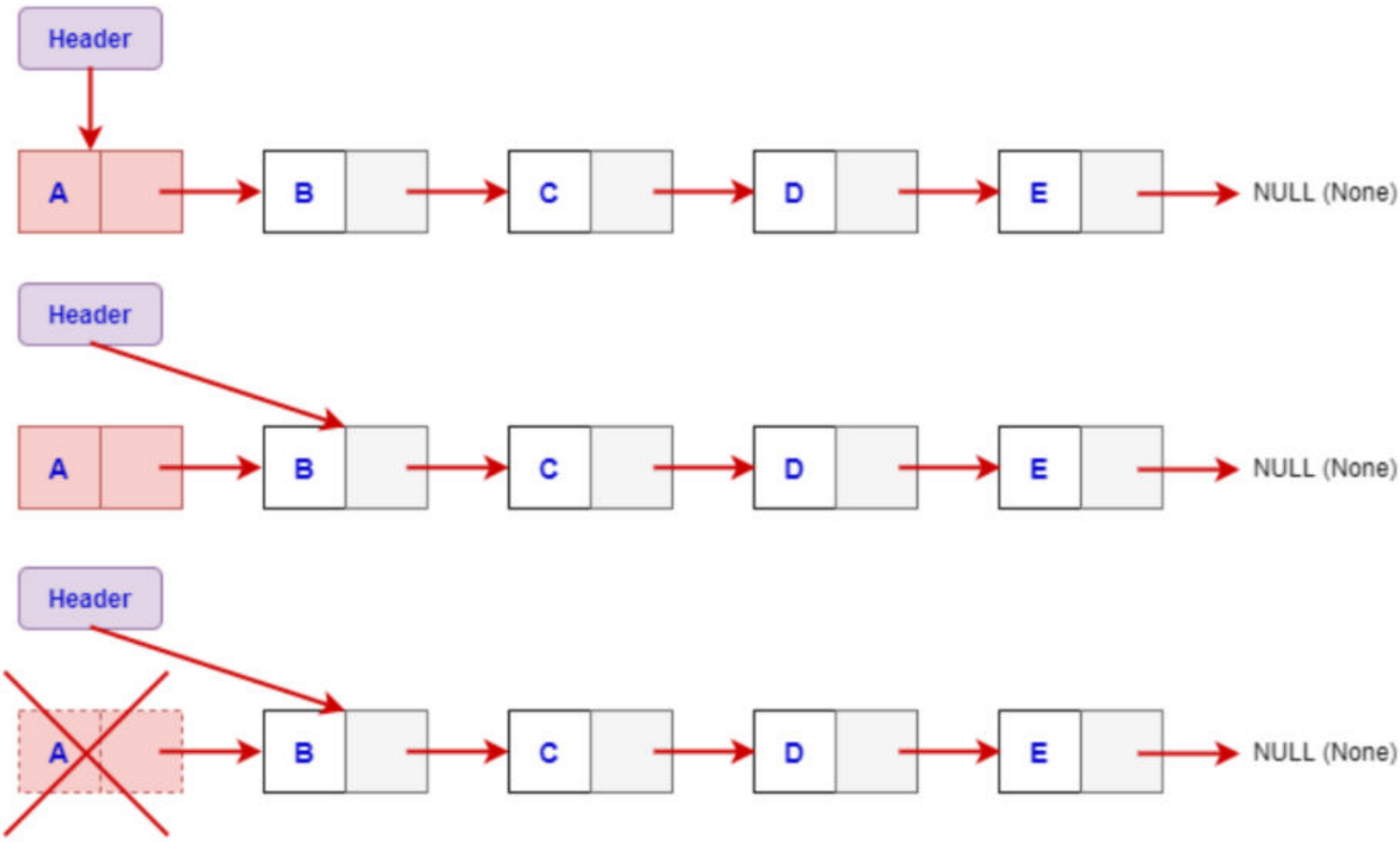
- Header is a link to the first node of the linked list
- **Flexible in size**
  - Size of a linked list grows or shrinks as and when new elements are inserted or deleted
- Random access is not possible in linked lists
  - The elements will have to be access sequentially
- Insertion or Removal of an element at/from any arbitrary position is efficient as none of the elements are required to be moved to new locations
  - Insertion at front

```
1. NewNode.Link = Header
2. Header = NewNode
```



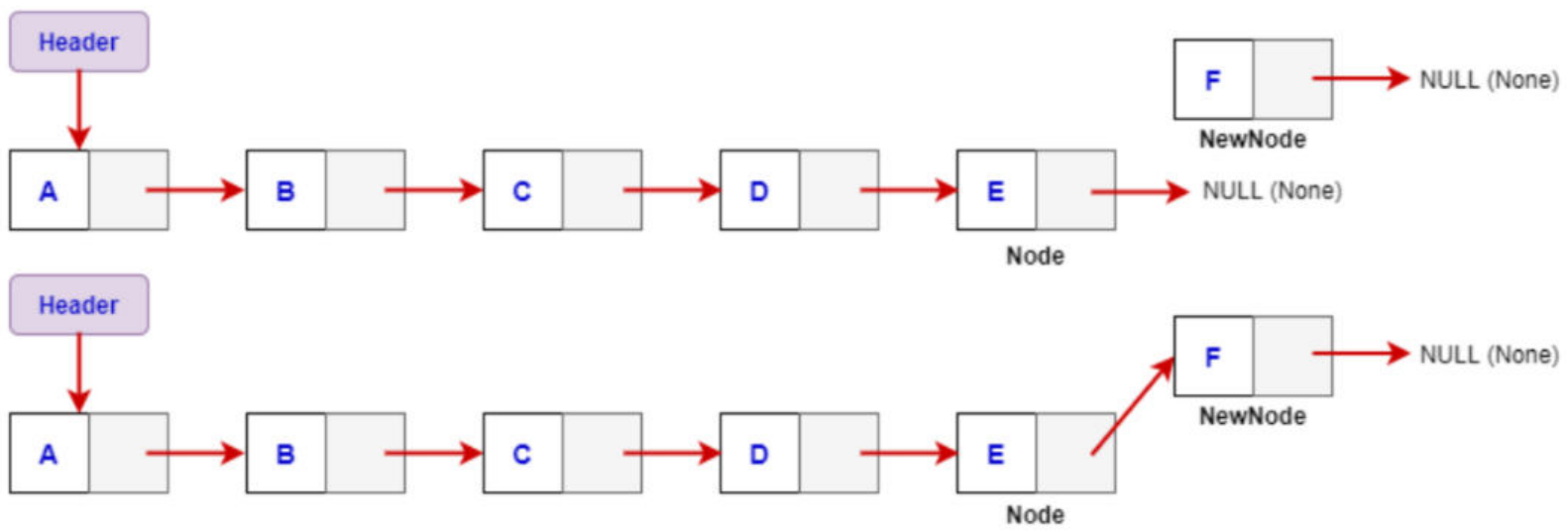
  - Remove from the front

```
1. Temp = Header
2. Header = Header.Link
3. Delete(Temp)
```



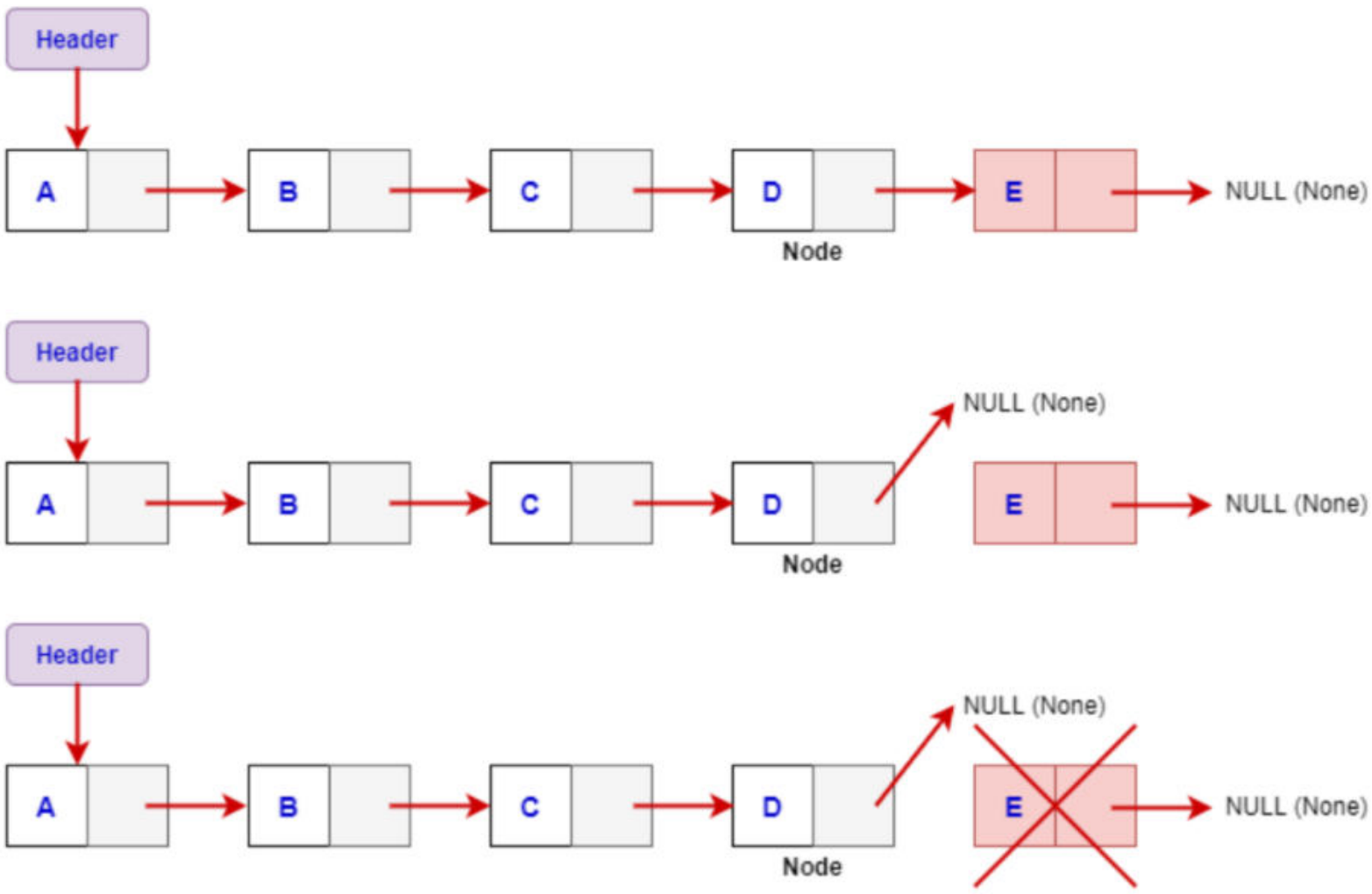
- Insertion at end

```
1. Node.Link = NewNode
```



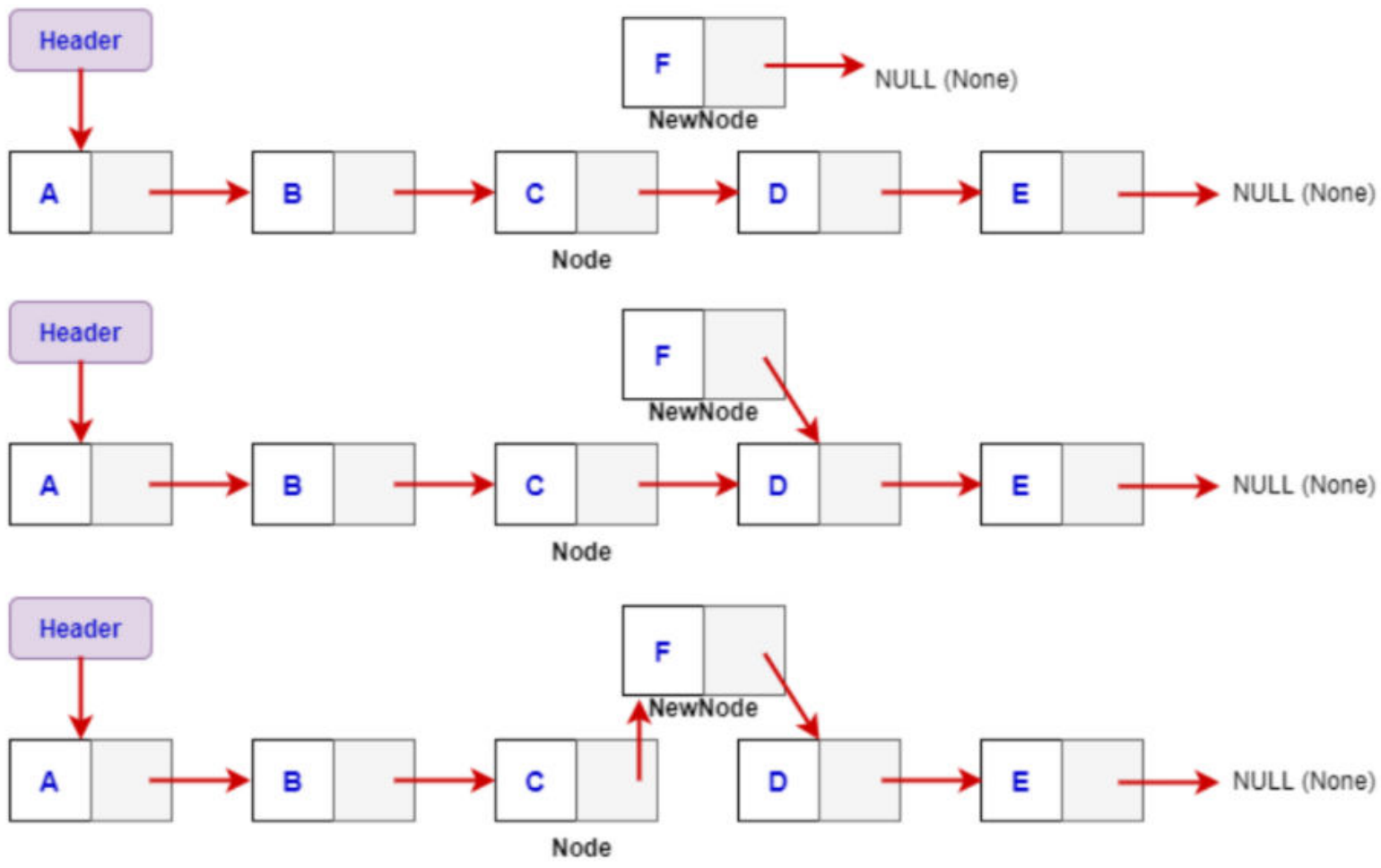
- Remove from end

```
1. Temp = Node.Link
2. Node.Link = NULL
3. Delete(Temp)
```



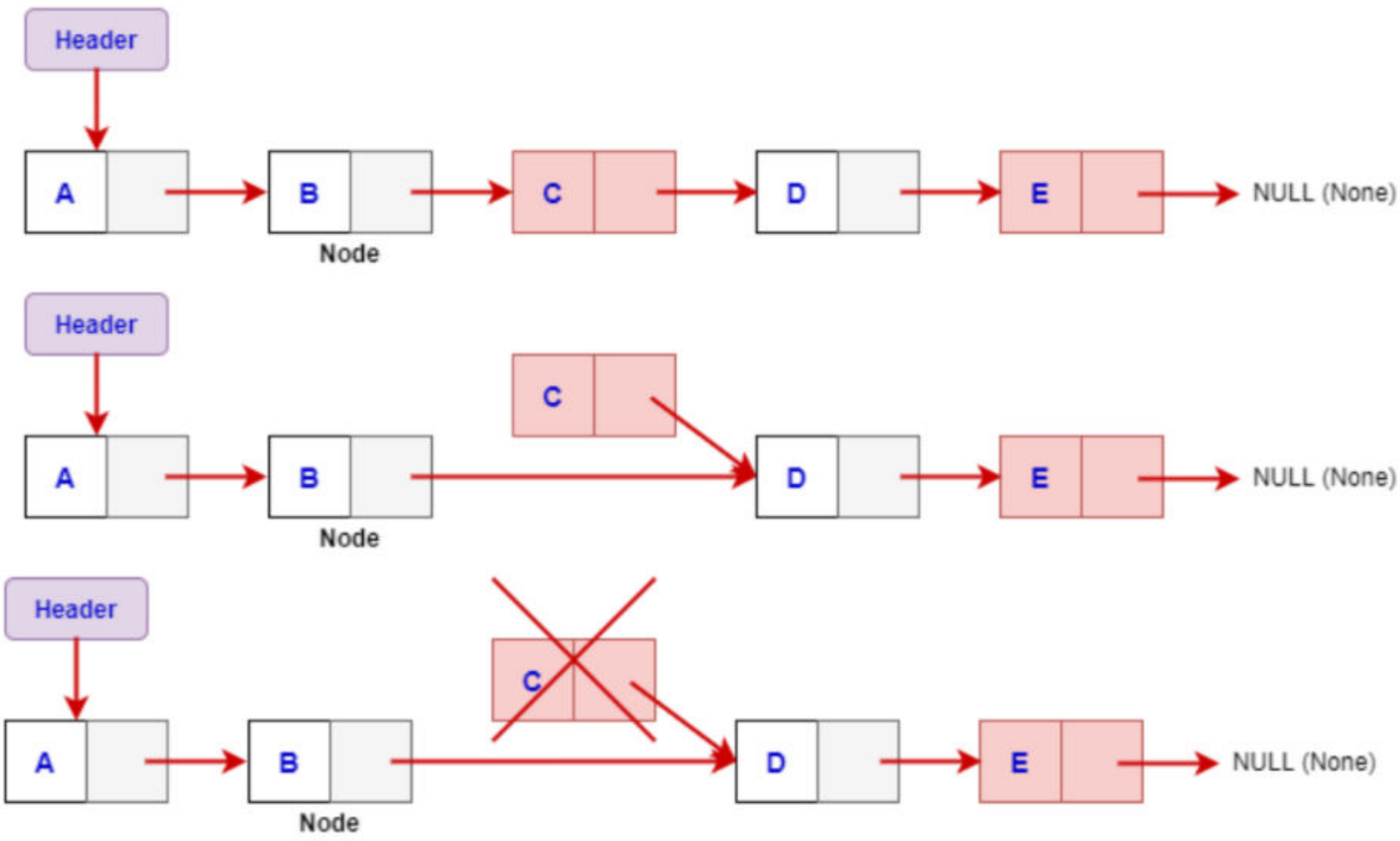
- Insertion at any intermediate position

```
1. NewNode.Link = Node.Link
2. Node.Link = NewNode
```



- Remove from any intermediate position

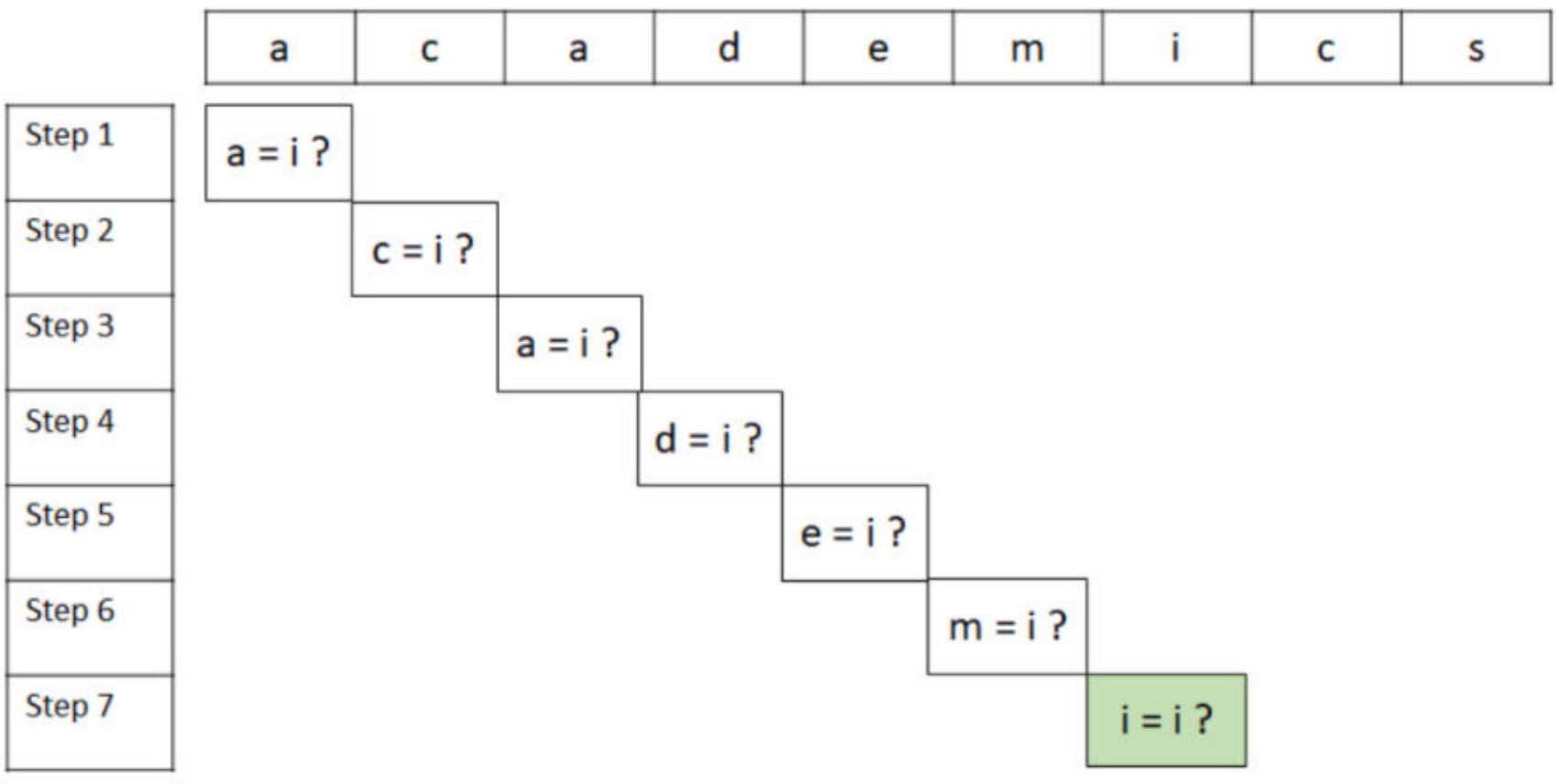
1. Temp = Node.Link
2. Node.Link = Node.Link.Link
3. Delete(Temp)



Linear Search

- The algorithm starts with the first element, compares with the given key value and returns yes if a match is found
- If it does not match, then it proceeds sequentially comparing each element of the list with the given key until a match has been found or the full list is traversed

Let the given input list be `inputArr = ['a', 'c', 'a', 'd', 'e', 'm', 'i', 'c', 's']` and the search key be `'i'`



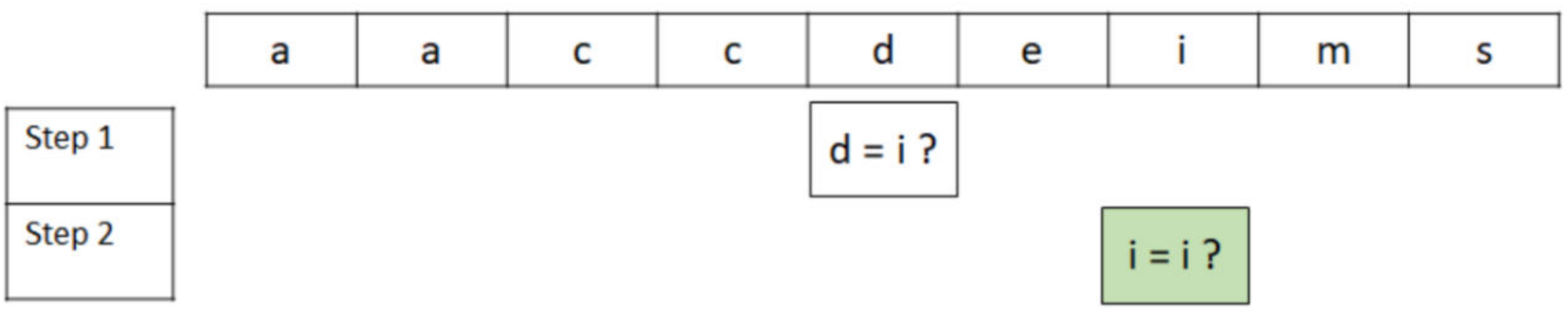

```
def linear_search(input_array, k):
  for i in range(len(input_array)):
    if input_array[i] == k:
      return i
  return -1

inputArr = ['a', 'c', 'a', 'd', 'e', 'm', 'i', 'c', 's']
k = 'i'
```

```
index = linear_search(inputArr, k)
if index != -1:
  print("Element found at " + index)
```

Binary Search

- The input for the algorithm is always a sorted list
- The algorithm compares the key k with the middle element in the list
- If the key matches, then it returns the index
- If the key does not match and is greater than the middle element, then the new list is the list to the right of the middle element
- If the key does not match and is less than the middle elements, then the new list is the list to the left of the middle element

Let the given input list be `inputArr = ['a', 'a', 'c', 'c', 'd', 'e', 'i', 'm', 's']` and the search key be `'i'`




```
def binary_search(arr, k):
  low = 0
  high = len(arr) - 1
  mid = 0

  while low <= high:
    mid = (low + high) // 2

    if arr[mid] < k:
      low = mid + 1
    elif arr[mid] > k:
      high = mid - 1
    else:
      return mid

  return -1

inputArr = ['a', 'a', 'c', 'c', 'd', 'e', 'i', 'm', 's']
k = 'i'
index = binary_search(inputArr, k)

if index != -1:
  print("Element found at " + index)
```

Common Data Structure operations

	Data Structure	Time Complexity								Space Complexity
		Average				Worst				Worst
		Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Linear Data Structures	Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
	Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Non-Linear Data Structures	Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
	Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
	Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
	Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
	B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
	AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Source: https://www.bigocheatsheet.com

Week 8 Lecture 3

Class	BSCCS2001
Created	@October 25, 2021 11:26 AM
Materials	
Module #	38
Type	Lecture
Week #	8

Algorithms and Data Structures: Data Structures

Non-linear data structures

Non-linear data structures: Why?

- From the study of Linear Data Structures in the previous module, we can make the following summary observations:
  - All of them have the space complexity $O(n)$, which is optimal
    - However, the actual used space may be lower in array while linked list has an overhead of 100% (double)
  - All of them have complexities that are identical for Worst as well as Average case
  - All of them offer satisfactory complexity for some operations while being unsatisfactory on the others

	Array		**Linked List**	
	Unordered	Ordered	Unordered	Ordered
Access	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Insert	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Delete	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Search	$O(n)$	$O(\lg n)$	$O(n)$	$O(n)$

- Non-linear data structures can be used to trade-off between extremes and achieve a balanced good performance for all

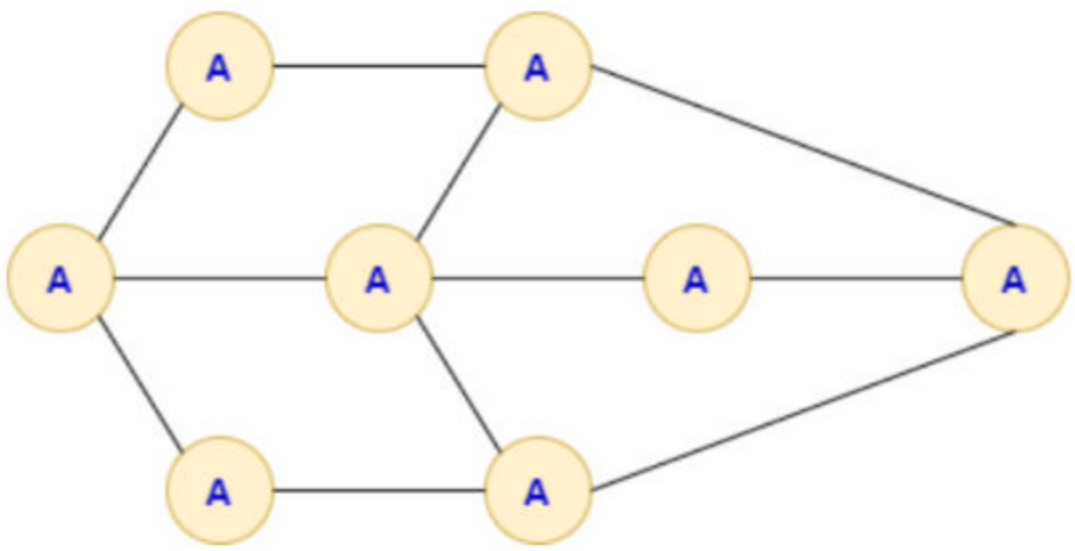
---

- Non-linear data structures are those data structures in which data items are not arranged in a sequence and each element may have multiple paths to connect to other elements
- Unlike linear data structures, in which each element is directly connected with utmost 2 neighbouring elements (previous and next elements), non-linear data structures may be connected with more than 2 elements
- The elements don't have a single path to connect to the other elements but have multiple parts
  - Traversing through the elements is not possible in one run as the data is non-linearly arranged
- Common Non-linear data structures include:
  - **Graph** → Undirected or Directed, Unweighted or Weighted and variants
  - **Tree** → Rooted or Unrooted, Binary or n-ary, Balance or Unbalanced and variants
  - **Hash Table** → Array with lists (coalesced chains) and one or more hash functions
  - **Skip List** → Multi-layered interconnected linked lists
  - and so on ...

Graph

- **Graph** → Graph G is a collection of vertices V (store the elements) and connecting edges (links) E between the vertices:

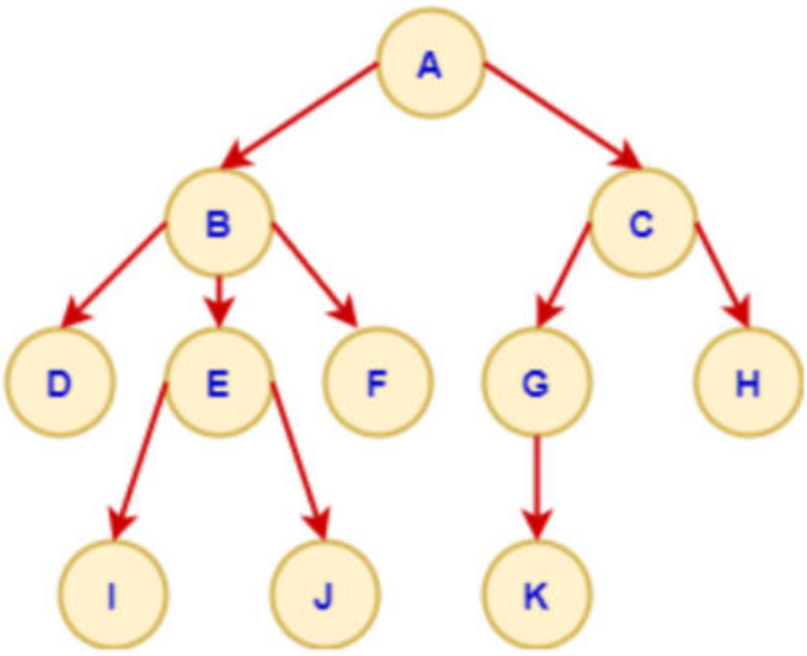
  $G =< V, E > \ where\ E \subseteq V \times V$



- A graph may be:
  - Undirected or Directed
  - Unweighted or Weighted
  - Cyclic or Acyclic
  - Disconnected or Connected
  - and so on ...
- Examples of a graph include
  - ER Diagram
  - Network: Electrical, Water
  - Friendships in Facebook
  - Knowledge Graph

Tree

- **Tree** → Is a connected acyclic graph representing hierarchical relationship



- A tree may be:
  - Rooted or Unrooted
  - Binary or n-ary
  - Balance or Unbalanced
  - Disconnected (forest) or Connected
  - and so on ...
- Examples of tree include:
  - Composite Attributes
  - Family Genealogy
  - Search Trees
  - and so on ...

- **Root** → The node at the top of the tree is called the root
  - There is only one root per tree and one path from the root node to any node
  - **A** is the root node in the given tree
- **Parent** → The node which is a predecessor of any node is called the parent node
  - In the given tree, **B** is the parent of **E**
  - Every node, except the root node, has a unique parent
- **Child** → A node which is the descendant of a node
  - **D**, **E** and **F** are the child nodes of **B**
- **Leaf** → A node which does not have any child node
  - **I**, **J** and **K** are leaf nodes

- **Internal Nodes** → The node which has at least one child is called the Internal Node
- **Sub-tree** → Sub-tree represents the tree rooted at that node
- **Path** → Path refers to the sequence of nodes along the edges of a tree
- **Siblings** → Nodes having the same parent
  - **D, E** and **F** are siblings
- **Arity** → Number of children of a node
  - **B** has arity 3, **E** has arity 2, **G** has arity 1 and **D** has arity 0 (Leaf)

  Maximum arity of a node is defined as the arity of the tree

---

- **Levels** → The root node is said to be at level 0 and the children of the root node are at level 1 and the children of the nodes which are at level 1 will be at level 2 and so on ...

Level is the length of the path (number of links) or distance of a node from the root node

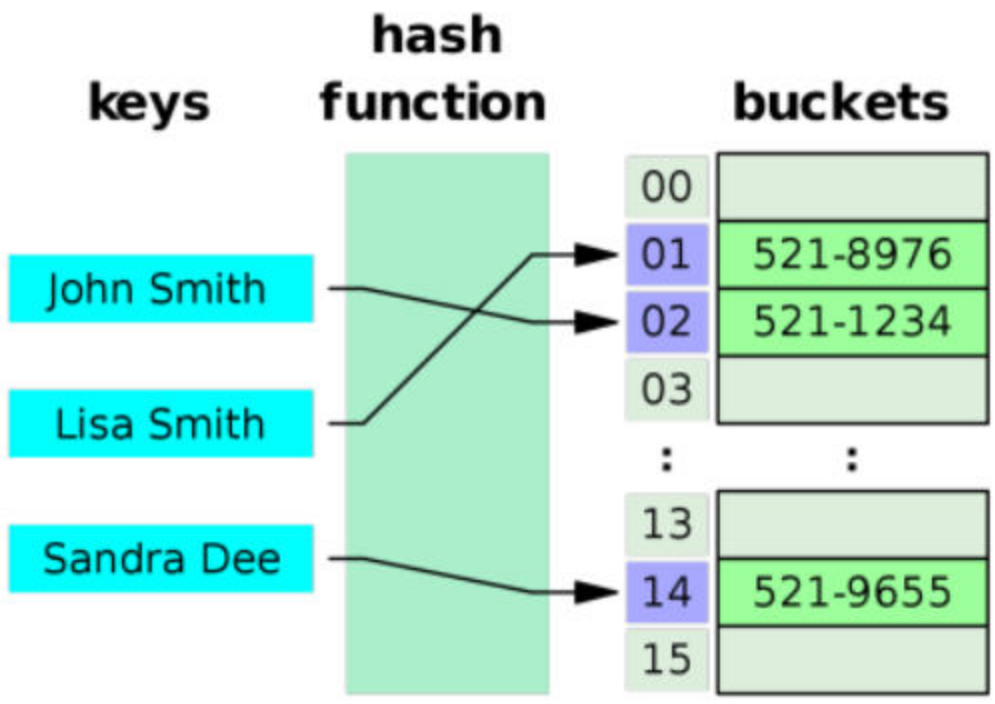
So, level of **A** is 0, level of **C** is 1, level of **G** is 2 and the level of **J** is 3

- **Height** → Max level in a tree
- **Binary Tree** → It is a tree, where each node can have at most 2 children
  - It has arity 2

- **Fact 1** → A tree with n nodes has $n-1$ edges
- **Fact 2** → The maximum number of nodes at level l of a binary tree is 2^l
- **Fact 3** → If h is the height of a binary tree of n nodes, then:
  - $h+1 \leq n \leq 2^{h+1}-1$
  - $\lceil lg(n+1) \rceil - 1 \leq h \leq n-1$
  - $O(lg\ n) \leq h \leq O(n)$
  - For a k-ary tree ,$O(lg_k\ n) \leq h \leq O(n)$

Hash Table

- **Hash Table (or Hash Map)** → Implements an associative array abstract data type, a structure that can map keys to values by using a hash function to compute an index (hash code), into an array of buckets or slots, from which the desired value can be found



- A hash table may be using:
  - Static or Dynamic Schemes
  - Open Addressing
  - 2-Choice Hashing
  - and so on ...
- Examples of Hash Tables include:
  - Associative arrays
  - Database indexing
  - Caches
  - and so on ...

Binary Search Tree

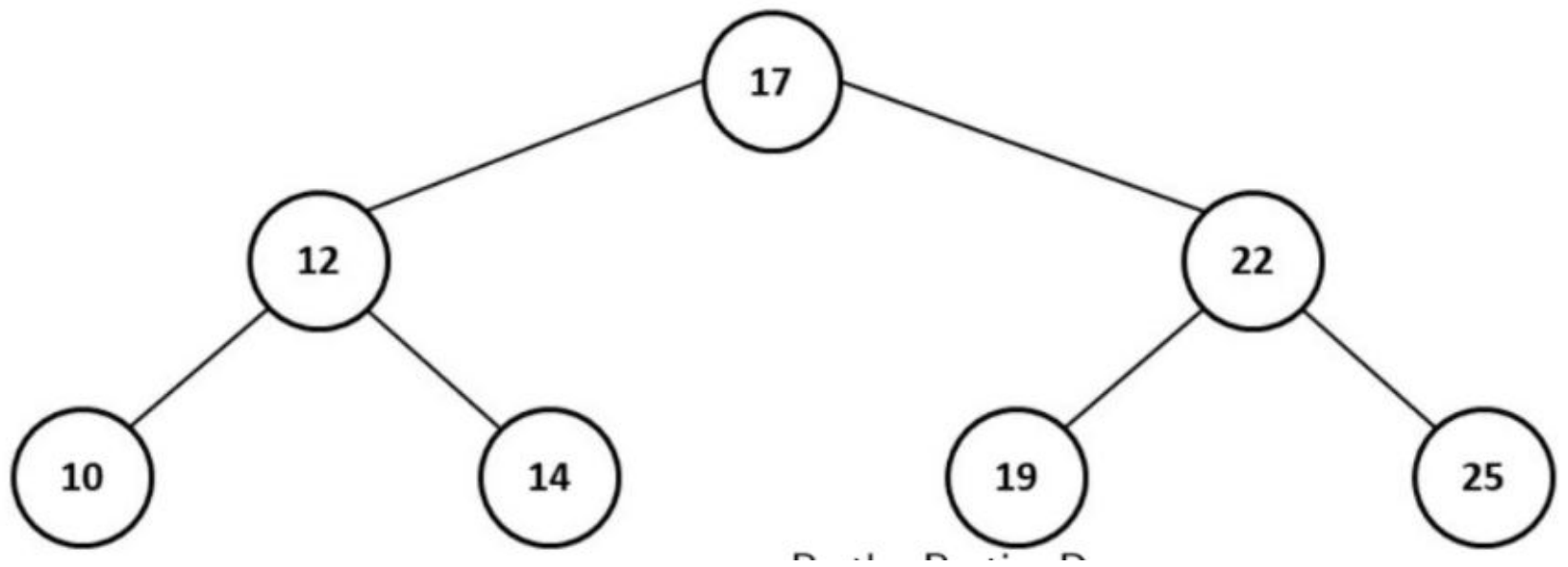
- During the study of linear data structure, we observed that
  - Binary search is efficient in the search of a key: $O(log\ n)$
    - It needs to be performed on a sorted array

    - The array makes insertion and deletion expensive at $O(n)$
  - The linked list, on the other hand, is efficient in insertion and deletion at $O(1)$, while it makes the search expensive at $O(n)$
    - $O(1)$ insert/delete is possible because we just need to manipulate the pointers and not physically move the data
- Using the non-linearity, specifically (binary) trees, we can combine the benefits of both
- Note that once an array is sorted, we know the order in which its elements may be checked (for any key) during a search
- As the binary search splits the array, we can conceptually consider the ***Middle Element*** to be the ***Root*** of a tree and ***left (right) sub-array*** to be its ***left (right) sub-tree***
- Progressing recursively, we have a **Binary Search Tree (BST)**

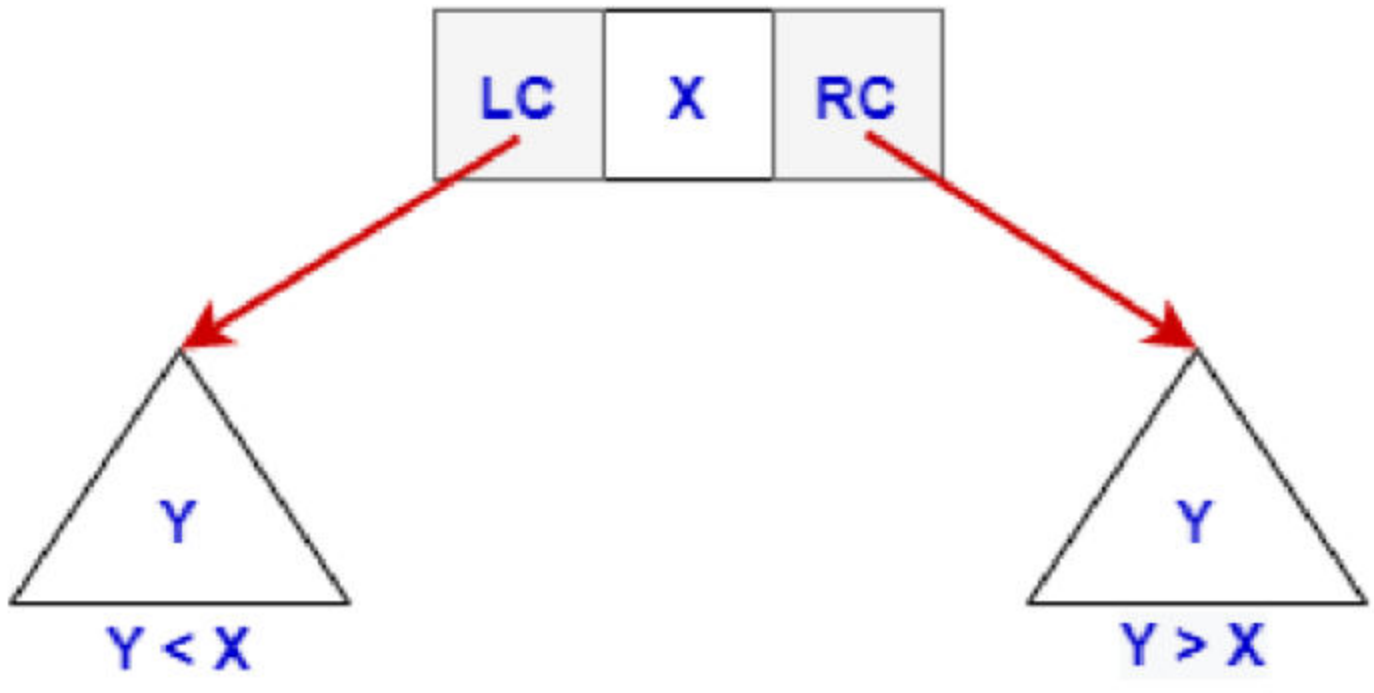
- Consider the data set:

10	12	14	17	19	22	25
LL	L	LR	M	RL	R	RR

- Search order is:
  - First → M
  - Second → L or R
  - Third:
    - For L → LL or LR
    - For R → RL or RR
  - Recursive ...
- Put as a tree



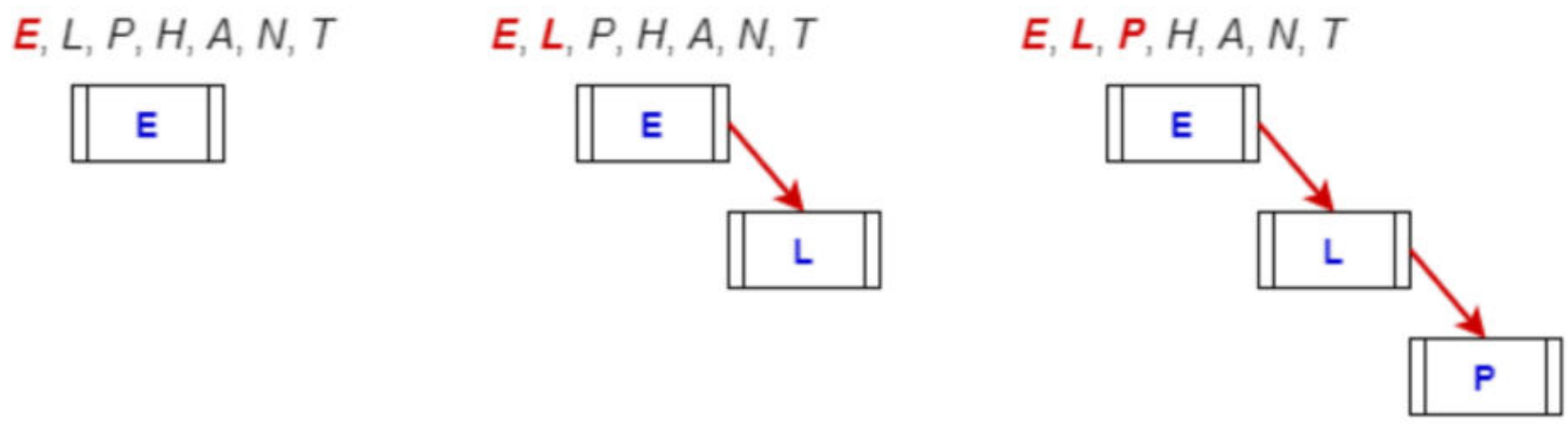
- **Binary Search Tree (BST)** → It is a tree in which all the nodes hold the following:
  - The value of each node in the left subtree is less than the value of its root
  - The value of each node in the right subtree is greater than the value of its root

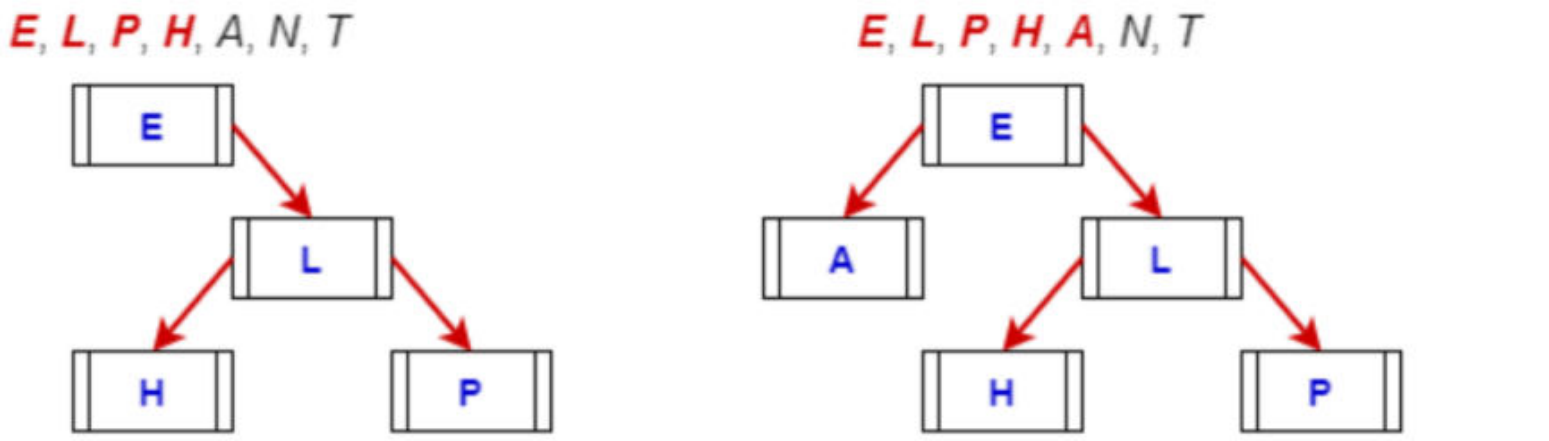


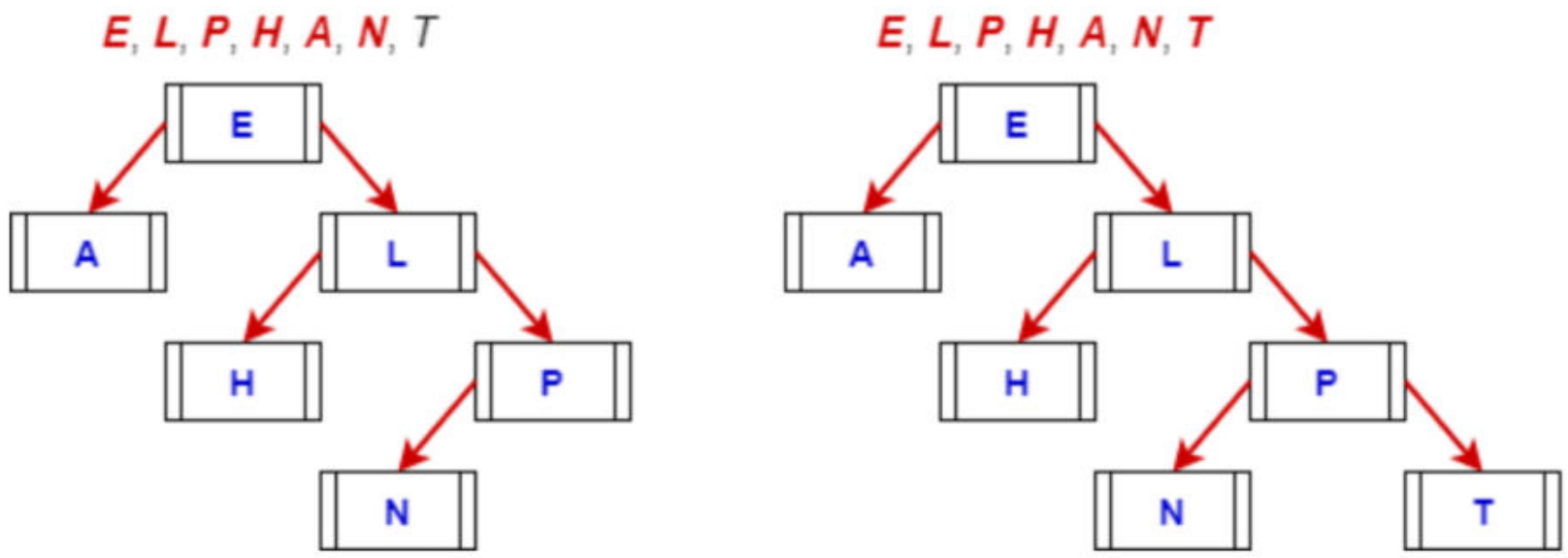
- **Structure of BST node** → Each node consists of an element (**X**), and a link to the left child or the left subtree (**LC**), and a link to the right child or the right subtree (**RC**)

- **Example** → Obtain the BST by inserting the following values:

  E, L, P, H, A, N, T



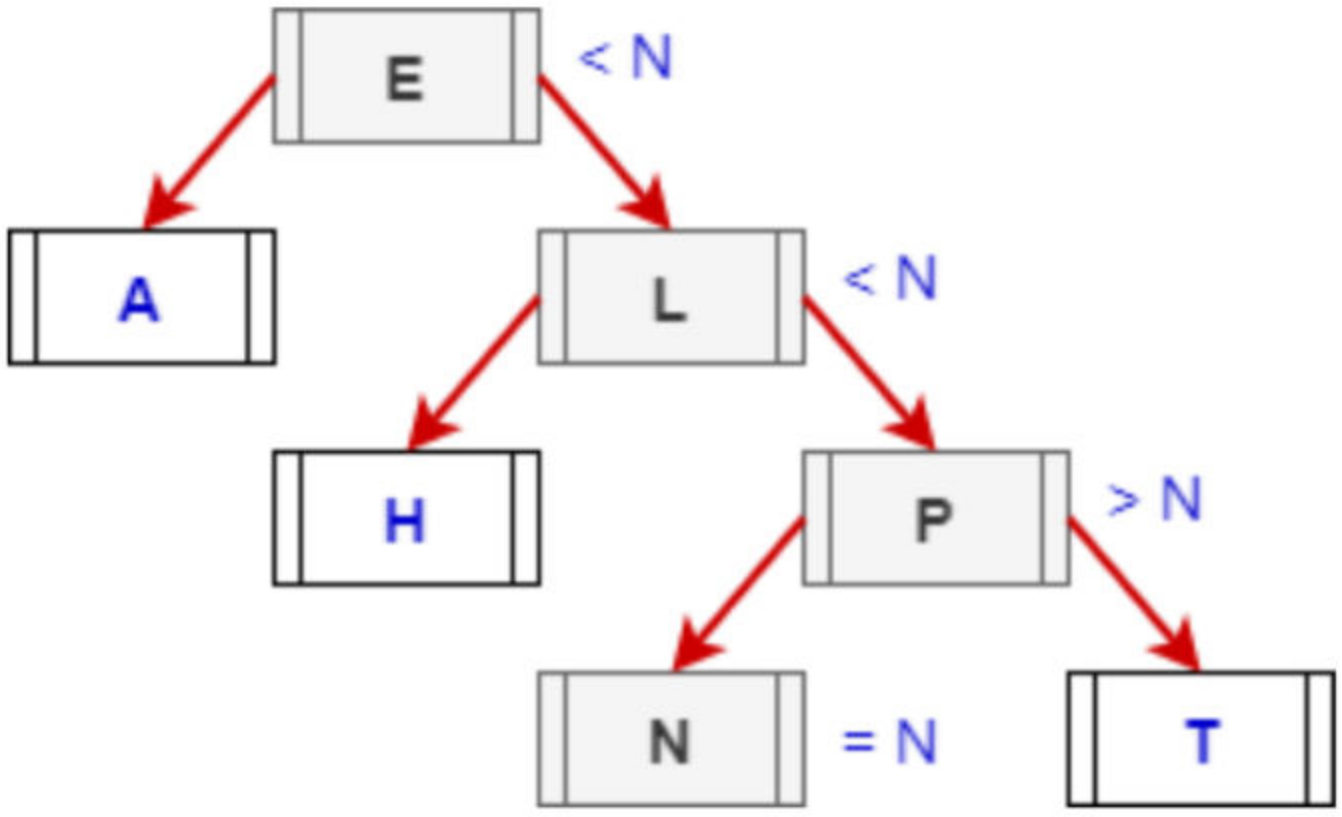




Searching a key in the BST

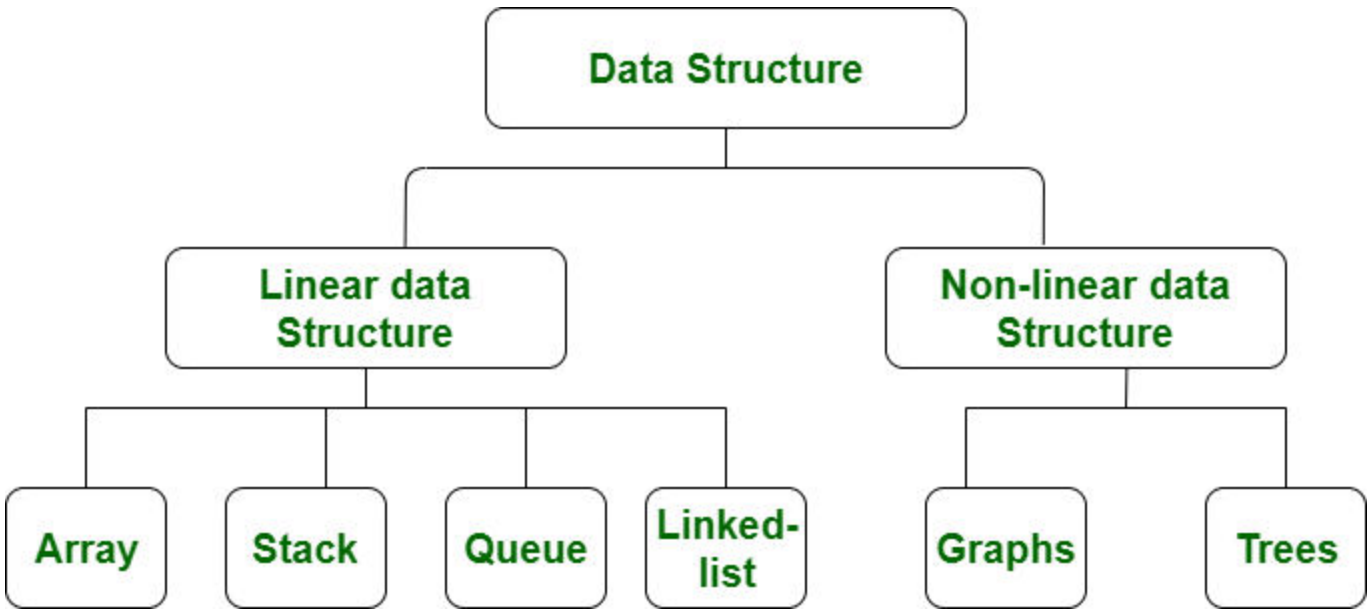
`search(root, key)`

1. Compare the key with the element at root
    a. If the key is equal to root's element then
        i. Element found and return
    b. else if they key is lesser than the root's element
        i. `search(root.lc)` # Search on the left subtree
    c. else: `(the key is greater than the root's element)`
        i. `search(root.rc)` # Search on the right subtree



- Searching a key in the BST is $O(h)$, where h is the height of the key
- **Worst Case**
    - The BST is skewed binary search tree (all the nodes except the leaf would have one child)
    - This can happen if they keys are inserted in sorted order
    - Height (h) of the BST having n elements becomes $n - 1$
    - Time complexity of search in BST becomes $O(n)$
- **Best Case**
    - The BST is balanced binary search tree
    - This is possible if
        - If the keys are inserted in purely randomized order
        - If the tree is explicitly balanced after every insertion
    - Height (h) of the binary search tree becomes $log\ n$
    - Time complexity of search in BST becomes $O(log\ n)$

Linear and non-Linear Data Structures



Source: https://www.geeksforgeeks.org/difference-between-linear-and-non-linear-data-structures/

Linear Data Structure	Non-linear Data Structure
In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.
In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.

Week 8 Lecture 4

Class	BSCCS2001
Created	@October 25, 2021 1:02 PM
Materials	
Module #	39
Type	Lecture
Week #	8

Storage and File Structure: Physical Storage

Classification of Physical Storage Media

- Speed with which data can be accessed
- Cost per unit of data
- Reliability
    - Data loss on power failure or system crash
    - Physical failure of the storage device
- Can differentiate storage into:
    - **Volatile storage** → Loses content when power is switched off
    - **Non-volatile storage** →
        - Content persists even when the power is off
        - Includes secondary and tertiary storage, as well as battery backed-up main memory

Physical Storage Media

- **Cache**
    - Fastest and most costly form of storage
    - Volatile
    - Managed by the computer system hardware
- **Main memory**

    - Fast access (10's to 100's of nanoseconds (ns))
        - 1 ns $= 10^{-9}$ seconds
    - Generally too small (or too expensive) to store the entire DB
        - Capacities of up to a few gigabytes widely used currently
        - Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2-3 years)
- **Volatile**
    - Contents of the main memory are usually lost if a power failure or system crash occurs

Physical Storage Media: Flash Memory

- Data survives power failure
- Data can be written at a location only once, but location can be erased and written to again
    - Can support only a limited number $(10K - 1M)$ of write/erase cycles
    - Erasing of memory has to be done to an entire bank of memory
- Reads are roughly as fast as main memory
- But writes are slow (few microseconds), erase is slower
- Widely used in embedded devices such as digital cameras, phones and USB keys

Physical Storage Media: Magnetic Disk

- Data is stored on spinning disk, and read/write magnetically
- Primary medium for the long-term storage of data
    - Typically stores the entire DB
- Data must be moved from the disk to main memory for access, and written back for storage — much slower access than the main memory
- Direct-access
    - Possible to read data on the disk in any order, unlike magnetic tape
- Capacities range up to roughly 16-32TB
    - Much larger capacity and much lower cost/byte than the main memory/flash memory
    - Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years)
- Survives power failures and system crashes
    - Disk failure can destroy data, but is rare

Physical Storage Media: Optical Storage

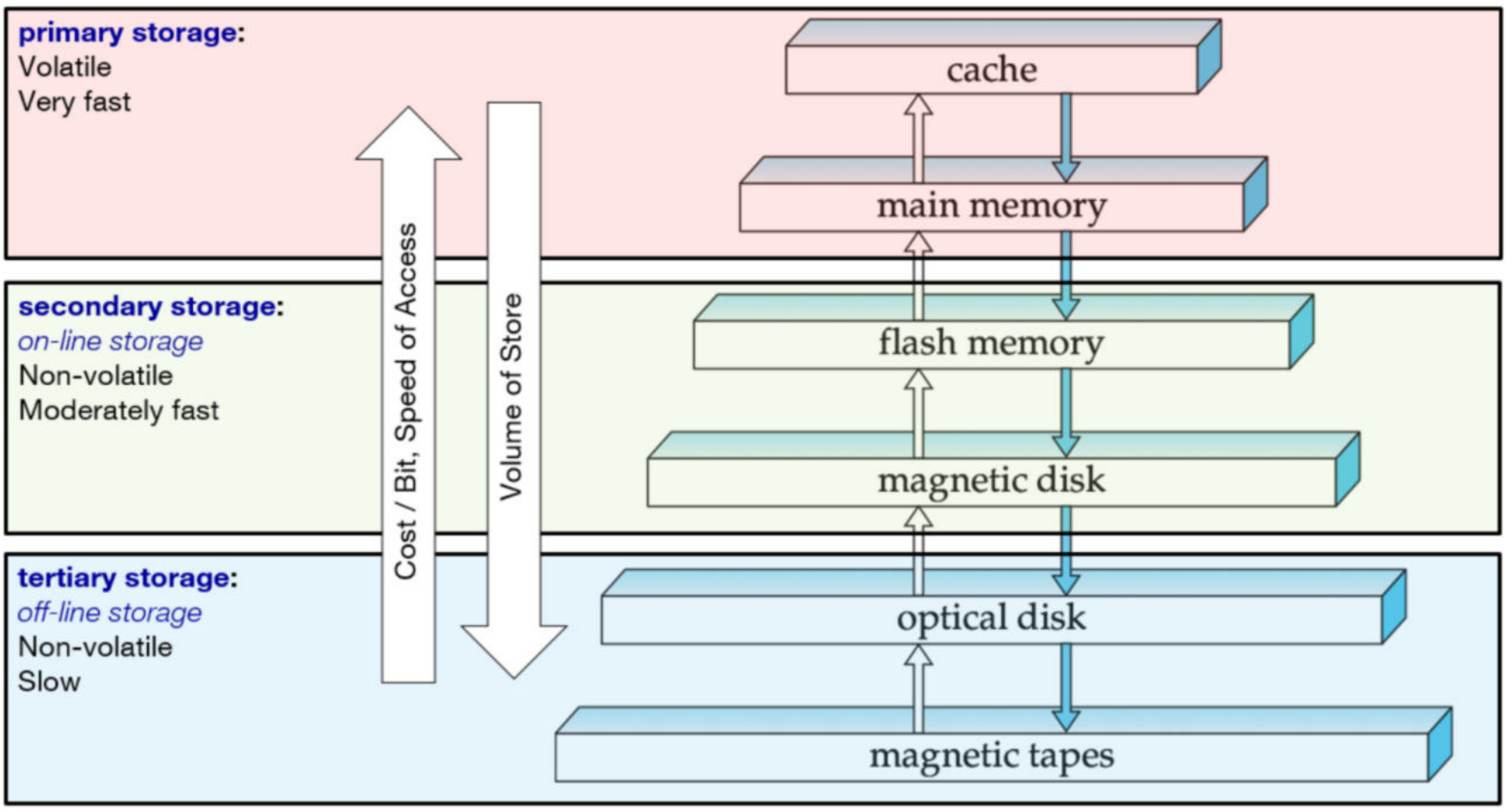
- Non-volatile, data is read optically from a spinning disk using a LASER
- CD-ROM (640MB) and DVD(4.7 to 17GB) most popular forms
- Blu-ray disks: 27GB to 54GB
- Write-one, Read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
- Multiple write versions also available (CD-RW, DVD-RW, DVD+RW and DVD-RAM)
- Reads and Writes are slower than with magnetic disks
- ***Juke-box*** systems, with large number of removable disks, a few drives and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

Physical Storage Media: Tape Storage

- Non-volatile, used primarily for backup (to recover from a disk failure) and for archival data
- **Sequential-access**
    - Much slower than a disk

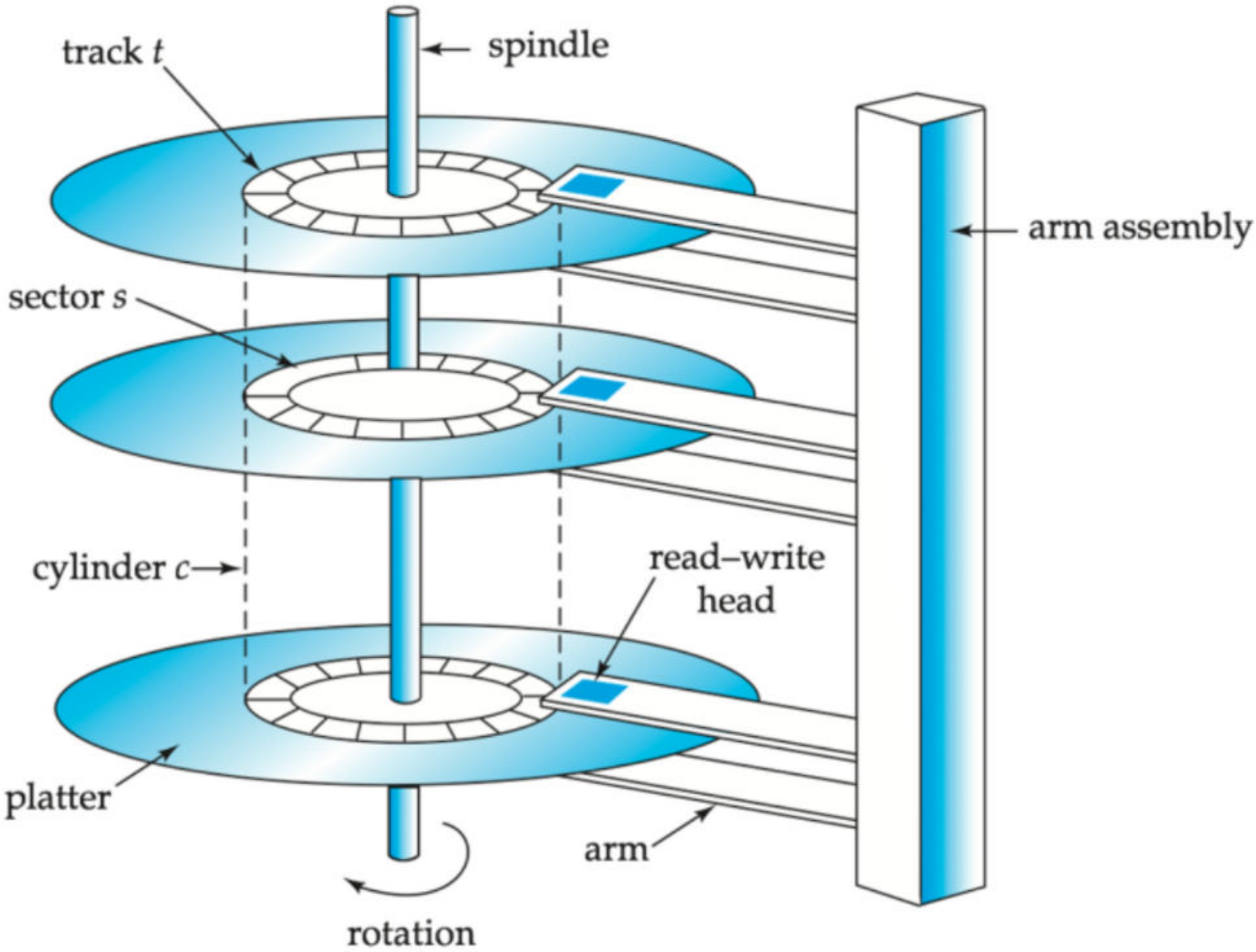
- Very high capacity (40-300TB tapes available)
- Tape can be removed from drive storage costs much cheaper than the disk, but drives are expensive
- Tape jukeboxes available for storing massive amounts of data
  - Hundreds of terabytes (TB) $(1TB = 10^{12}\ bytes)$ to even multiple petabytes (PB) $(1PB = 10^{15}\ bytes)$

Storage hierarchy



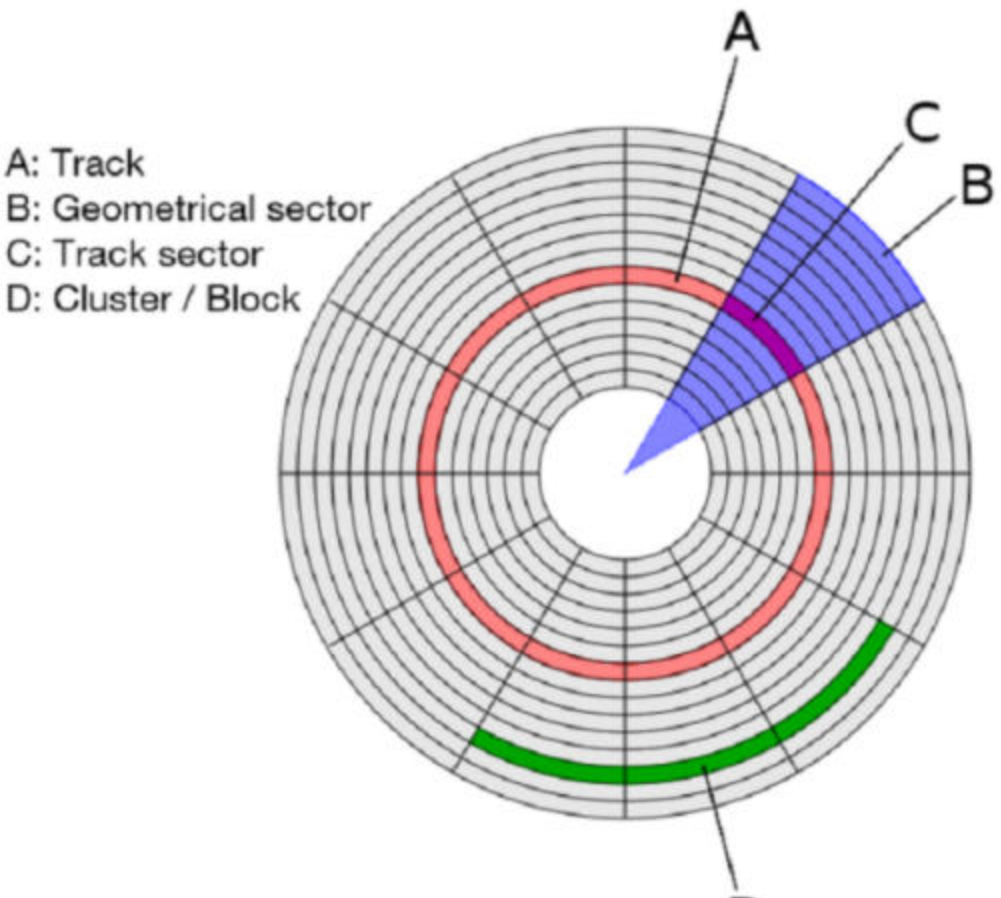
Magnetic Disk

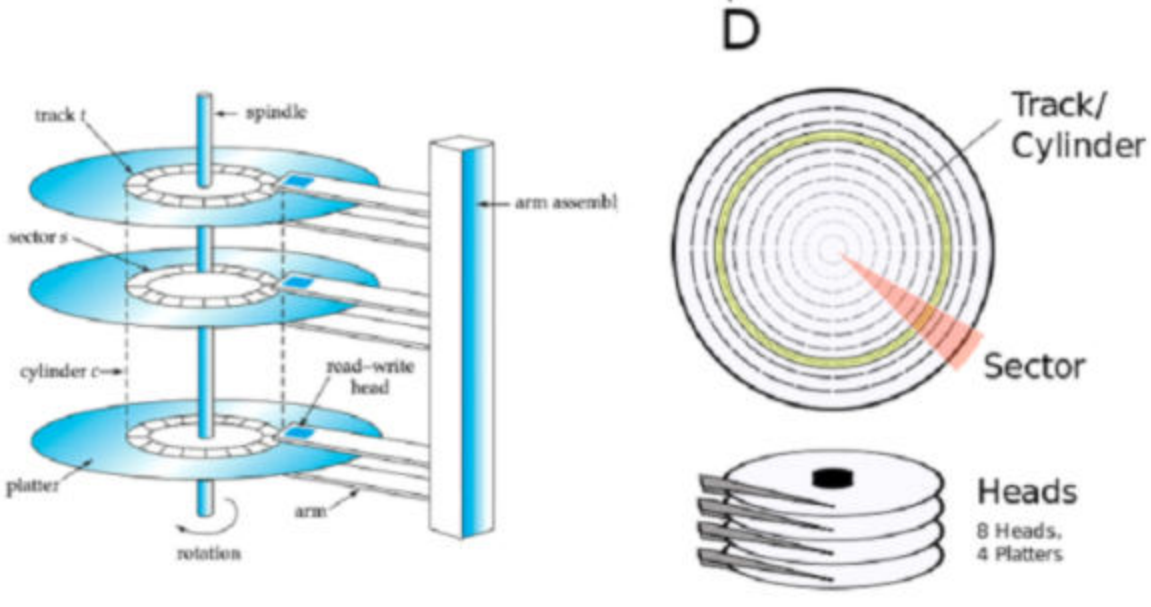
Magnetic Disk: Mechanism



NOTE: Diagram is schematic, and simplifies the structure of actual disk drives

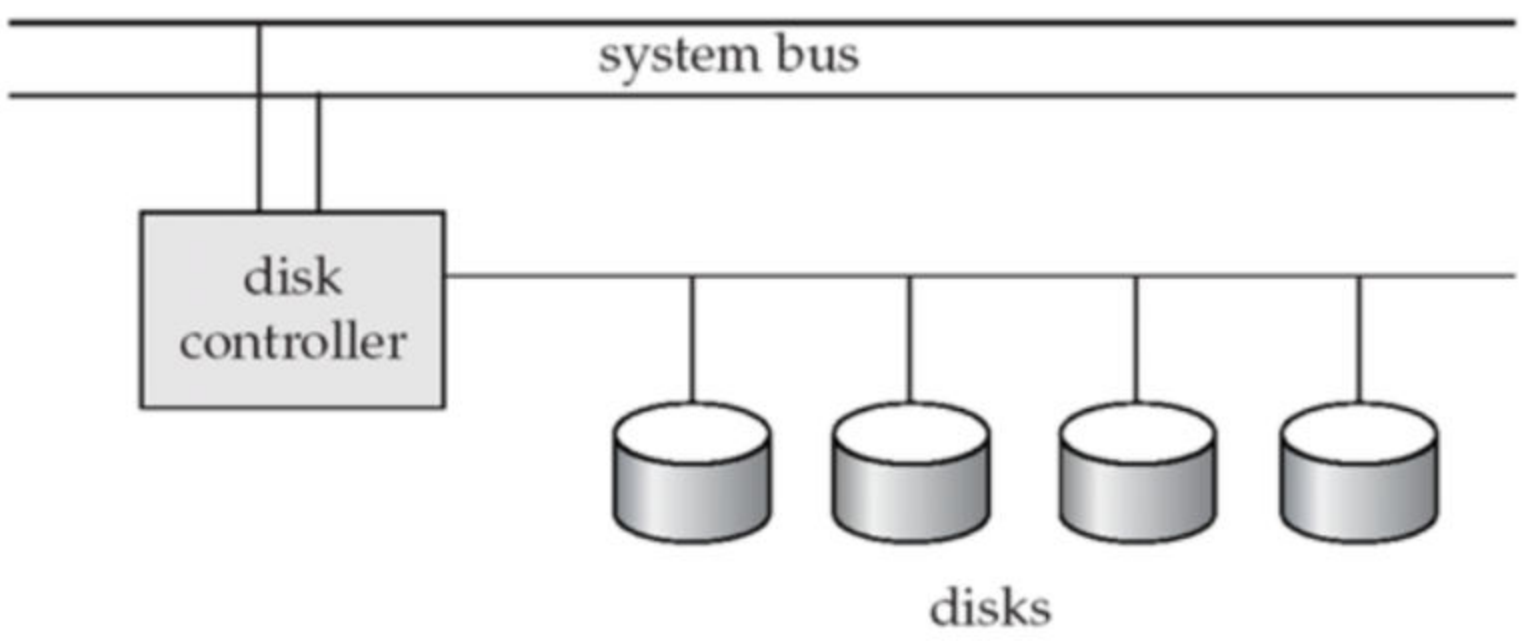
- Read-write head
  - Positioned very close to the platter surface
- Surface of the platter is divided into circular ***tracks***
  - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into ***sectors***
  - A sector is the smallest unit of data read or written
  - Sector size typically 512 bytes
  - Sectors / track → 500 to 1K (inner) to 1K to 2K (outer)
- To read/write a sector
  - Disk arm swings to position head on right track
  - Platter spins → Read/Write as sector passes under the head
- Head-disk assemblies
  - Multiple disk platters on a single spindle (1 to 5 usually)
  - One head per platter, mounted on a common arm
- Cylinder *i* consists of i^{th} track of all the platters





Magnetic Disk: Disk Controller, Subsystems and Interfaces

- **Disk Controller** → Interfaces between the computer system and the disk drive hardware
  - Accepts high-level commands to read or write a sector
  - Initiates actions moving the disk arm to the right track, reading or writing the data
  - Computes and attaches checksums to each sector to verify that correct read back
  - Ensures successful writing by reading back sector after writing it
  - Performs remapping of bad sectors
- **Disk Subsystem**



- **Disk Interface Standards Families** → ATA, SATA, SCSI, SAS and several variants
- **Storage Area Networks (SAN)** → Connects disks by a high-speed network to a number of servers
- **Network Attached Storage (NAS)** → Provides a file system interface using networked file system protocol

Magnetic Disk: Performance Measures

- **Access Time** → Time from a read or write request issue to start of data transfer
  - **Seek Time** → Time to reposition the arm over the correct track
    - Avg. seek time is 1/2 the worst case seek time; 1/3 if all the tracks have same number of sectors
    - 4 to 10 milliseconds on typical disks
  - **Rotational Latency** → Time for the sector to be accessed to appear under the head
    - Average latency is 1/2 of the worst case latency
    - 4 to 11 milliseconds on typical disks (5400 to 15000 RPM)
- **Data-transfer rate** → The rate at which data can be retrieved from or stored to the disk
  - 25 to 100MB per second max rate, lower for inner tracks
  - Multiple disks may share a controller, so rate that controller can handle is also important
- **Mean Time to Failure (MTTF)** → Avg. time the disk is expected to run continuously without any failure
  - Typically 3 to 5 years
  - Probability of failure of new disks is quite low, corresponding to a theoretical MTTF of 500,000 to 1,200,000 hours for a new disk
    - For example → an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
  - MTTF decreases as the disk ages

Magnetic Tapes

- Hold large volumes of data and provide high transfer rates
  - Tape formats
    - Few GB for DAT (Digital Audio Tape) format
    - 10 - 40GB with DLT (Digital Linear Tape) format
    - 100GB+ with Ultrium format
    - 330GB with Ampex helical scan format
  - Transfer rates from a few to 10's of MB/s
- Tapes are cheap, but the cost of drives is very high
- Very slow access time in comparison to magnetic and optical disks
  - Limited to sequential access
  - Some formats (Accelis) provide faster seek (10's of seconds) at cost of lower capacity
- Used mainly for backup, for storage of infrequently used information and as an offline medium for transferring information from one system to another
- Tape jukeboxes used for very large capacity storage
  - Multiple petabytes $(10^{15}\ bytes)$

Cloud Storage

- Cloud storage is purchased from a 3rd party cloud vendor who owns and operates the data storage capacity and delivers it over the internet in a pay-as-you-go model
- The cloud storage vendors manage capacity, security and durability to make data accessible to applications all around the world
- Applications access cloud storage through traditional storage protocols or directly via an API

- Many vendors offer complementary services designed to help collect, manage, secure and analyze data at a massive scale
    - Various available options for cloud storage are:
        - Google Drive
        - Amazon Drive
        - Microsoft Onedrive
        - Evernote
        - Dropbox
        - and so on ...

Cloud storage v/s Traditional Storage

Parameters	Cloud Storage	Traditional Storage
Cost	Cloud storage is cheaper per GB than using external drives.	The hardware and infrastructure costs are high and adding on more space and upgrading only adds extra costs.
Reliability	Cloud storage is highly reliable as it takes less time to get under functioning	Traditional storage requires high initial effort and is less reliable.
File Sharing	Cloud storage supports file sharing dynamically as it can be shared anywhere with network access	Traditional storage requires physical drives to share data and a network is to be established between both
Accessibility	Cloud storage gives you access to your files from anywhere	Restricted to local access
Backup/ Recovery	Very safe from on site disaster. In case of a hard drive failure or other hardware malfunction, you can access your files on the cloud, which acts as a backup solution for your local storage on physical drives	Data that is stored locally is much more susceptible to unexpected events and local storage and local backups could be easily lost

Other storage

Optical Disks

- Compact disk-read only memory (CD-ROM)
    - Removable disks, 640MB per disk
    - Seek time about 100 msec (optical read is heavier and slower)
    - Higher latency (3000 RPM) and lower data-transfer rates (3 - 6MB/s) compared to magnetic disks
- Digital Video Disk (DVD)
    - DVD-5 holds 4.7GB and DVD-9 holds 8.5GB
    - DVD-10 and DVD-18 are double sided formats with capacities of 9.4GB and 17GB
    - Blu-ray DVD → 27GB (54GB for double sided disk)
    - Slow seek time, for same reasons as CD-ROM
- Record once versions (CD-R and DVD-R) are popular
    - Data can only be written once and cannot be erased
    - High capacity and long lifetime; used for archival storage
    - Multi-write versions (CD-RW, DVD-RW, DVD+RW and DVD-RAM) also available

Flash drives

- Flash drives are often referred to as pen drives, thumb drives or jump drives
    - They have completely replaced floppy drives for portable storage

    - Considering how large and inexpensive they have become, they have also replaced CDs and DVDs for data storage purposes
- USB flash drives are removable and re-writable storage devices that, as the name suggests, require a USB port for connection and utilizes non-volatile flash memory technology
- The storage space in USB is quite large with sizes ranging from 128MB to 2TB
- The USB standard a flash drive is built around will determine the number of things about its potential performance, including maximum transfer rate

Secure Digital Cards (SD Cards)

- A Secure Digital (SD) card is a type of removable memory card used to read and write large quantities of data
- Due to their relatively small size, SD cards are widely used in mobile electronics, cameras, smart devices, video game consoles and more
- There are several types of SD cards sold and used today:

Card Type	Year of Debut	Capacity	Supported Devices
SD	1996	128MB to 2GB	All host devices that support SD, SDHC, SDXC
SDHC	2006	4GB to 32GB	All host devices that support SDHC, SDXC
SDXC	2009	64GB to 2TB	All host devices that support SDXC

Card Type	Capacity	File System	Remarks
SD	128MB to 2GB	FAT16	FAT16 supports 16 MB to 2 GB
SDHC	4GB to 32GB	FAT32	FAT32 can be support up to 16 TB
SDXC	64GB to 2TB	exFAT	exFAT is non-standard, supports file up to 4 GB

Source: https://integralmemory.com/faq/what-are-differences-between-fat16-fat32-and-exfat-file-systems

Flash storage

- NOR Flash vs NAND Flash
- NAND Flash
    - Used widely for storage, since it is much cheaper than NOR Flash
    - Requires page-at-a-time read (page: 515 bytes to 4KB)
    - Transfer rate around 20MB/s
    - **Solid State Disks** → Use multiple flash storage devices to provide higher transfer rate of 200MB/s or higher
    - Erase is very slow (1 to 2ms)
        - Erase block contains multiple pages
        - Remapping of logical page addresses to physical page addresses avoids waiting for erase
            - Translation table tracks mapping
            - Also stored in a label field of flash page
            - Remapping carried out by flash translation layer
        - After 100,000 to 1,000,000 erases, erase block becomes unreliable and cannot be used
            - Wear leveling

Solid State Drives (SSDs)

- SSDs replace traditional mechanical hard disks by using flash-based memory, which is significantly faster
- SSDs speed up computers significantly due to their low read-access time and fast throughput
- The idea of SSDs was introduced in 1978
    - It was implemented using semiconductors

    - It stores the data in the persistent state even when no power is supplied
- The speed of SSD much larger than that of HDD as it reads/writes data at a higher input-output per second
- Unlike HDDs, SSDs do not include any moving parts
    - SSDs resists vibrations and high temperatures

SSD v/s HDD

Parameters	SSD	HDD
Technology	Integrated circuit using Flash memory	Mechanical Parts, including spinning disks or platters
Access Time	0.1 ms	5.5-8.0 ms
Average Seek Time	0.08-0.16 ms	$<$ 10 ms
Speed (SATA II)	80-250 MB/sec	65-85 MB/sec
Random I/O Performance	6000 io/s	400 io/s
Backup rates	6 hours	20- 24 hours
Reliability	The failure rate of less than 0.5%	Failure rate fluctuates between 2-5%
Energy Consumption	2 to 5 watts	6 to 15 watts

Future of Storage

DNA Digital Storage

Ooooooh, we going Cyberpunk

- DNA digital data storage is the process of encoding and decoding binary data to and from synthesized strands of DNA
- While DNA as a storage medium has enormous potential because of its high storage density, its practical use is currently severely limited because of its high cost and very slow read and write times
- Digital storage systems encode the text, photos, videos or any kind of information as a series of 0s and 1s
    - This same information can be encoded in DNA using the 4 nucleotides that make up the genetic code: A, T, G and C
    - For example → G and C could be used to represent 0 while A and T represent 1
- DNA has several other features that makes it desirable as a storage medium; it is extremely stable and is fairly easy (but expensive) to synthesize and sequence
- Also, because of its high density — each nucleotide, equivalent to up to 2 bits, is about 1 cubic nanometer - an exabyte $(10^{18}\ bytes)$ of data stored as DNA could fit in the palm of our hands
- DNA synthesis → A DNA synthesizer machine builds synthetic DNA strands matching the sequence of digital code

Quantum Memory

- Quantum Memory is the quantum-mechanical version of ordinary computer memory
- Whereas ordinary memory stores information as binary states (represented by 1's and 0's)
    - Quantum memory stores a quantum state for later retrieval
- These states hold useful computation information knows as ***qubits***
- Quantum memory is essential for the development of many devices in quantum information processing applications such as quantum network, quantum repeater, linear optical quantum computation or long-distance quantum communication

- Unlike the classical memory of everyday computers, the states stored in quantum memory can be in a quantum superposition, giving much more practical flexibility in quantum algorithms than classical information storage

Week 8 Lecture 5

Class	BSCCS2001
Created	@October 25, 2021 2:53 PM
Materials	
Module #	40
Type	Lecture
Week #	8

Storage and File Structure: File Structure

File Organization

- A Database is
  - A collection of files
    - A file is
      - A sequence of records
        - A record is
          - A sequence of fields
- One approach:
  - Assume record size is fixed
  - Each file has records of one particular type only
  - Different files are used for different relations
  - This case is easiest to implement; will consider variable length records later
- A Database file is partitioned into fixed-length storage units called blocks
  - Blocks are units of both storage allocation and data transfer

Fixed-Length Records

- Simple approach
  - Store record i starting from byte $n * (i - 1)$, where n is the size of each record

- Record access is simple but records may cross blocks
  - Modification → Do not allow records to cross block boundaries

- Deletion of record i → Alternatives
  - Move records $i+1, ..., n$ to $i, ..., n-1$
  - Move record n to i
  - Do not move records, but link all the free records on a free list

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Deleting Record 3 with Comparison

Before deletion

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

After deletion & Compaction

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Deleting Record 3 with Moving last record

Before deletion

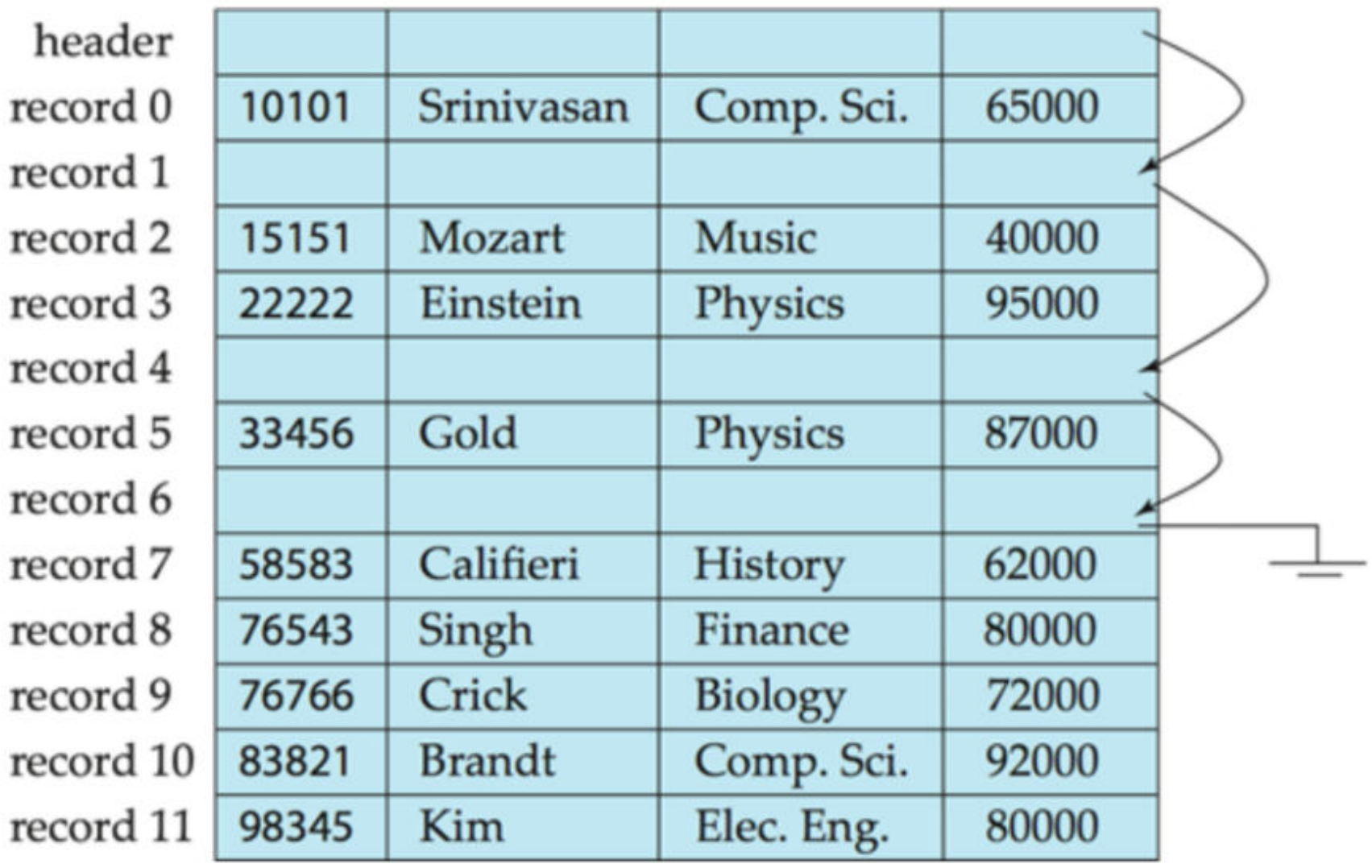
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

After deletion & Movement

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

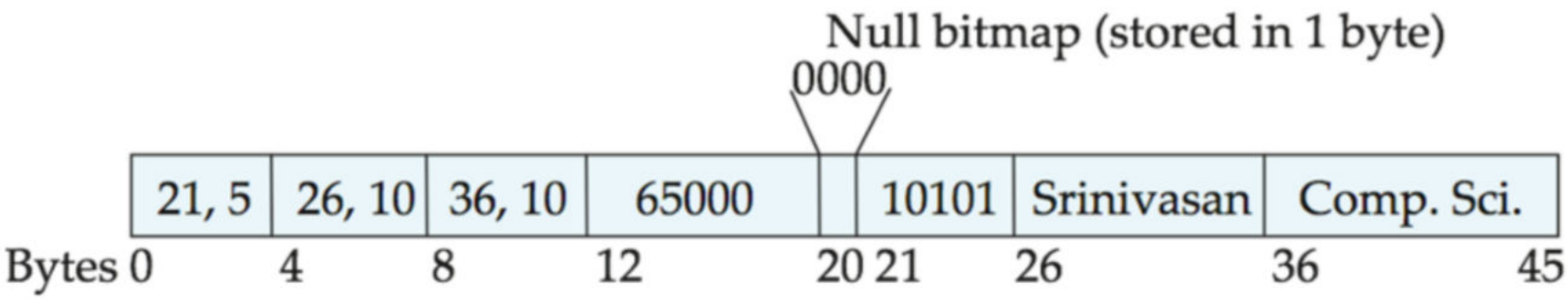
Free Lists

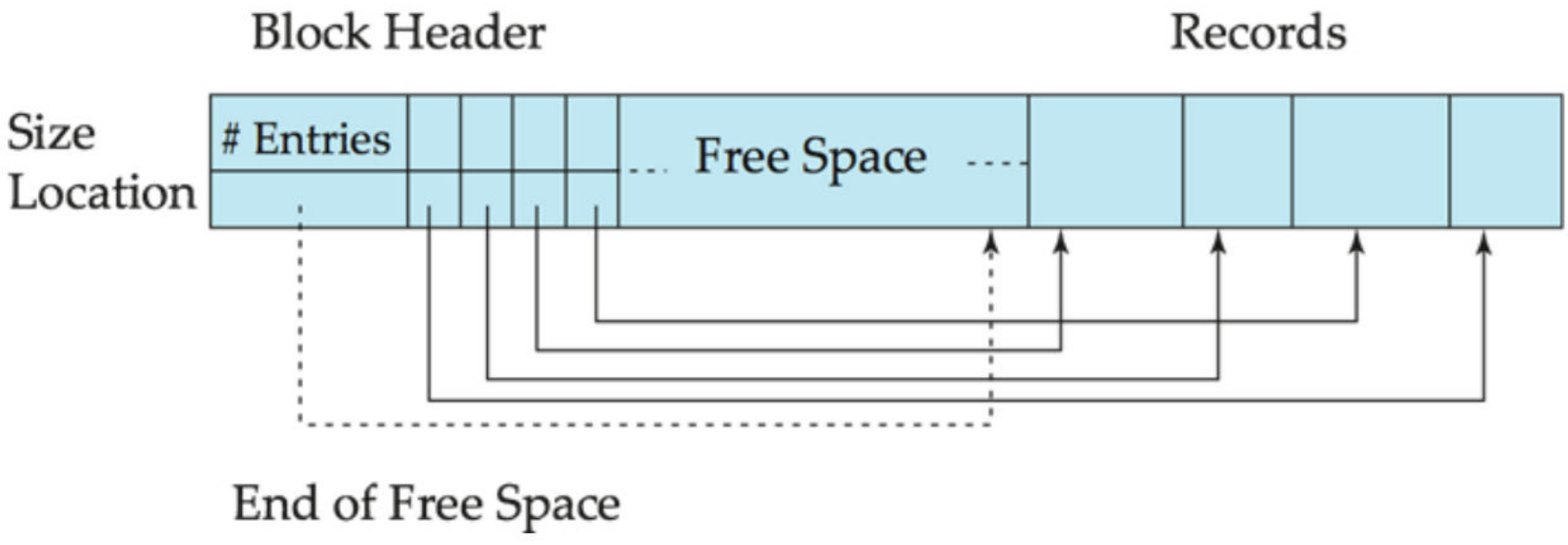
- Store the address of the first deleted record in the file header
- Use this first record to store the address of the second deleted record, and so on ...
- Consider these stored addresses as pointers since they point to the location of the record
- More space efficient representation → Re-use space for normal attributes of free records to store pointers (No pointers stored in in-use records)



Variable-Length Records

- Variable-length records arise in DB systems in several ways:
  - Storage of multiple record types in a file
  - Record types that allow variable lengths for one or more fields such as strings (varchar)
  - Record types that allow repeating fields (used in some older data models)
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length) with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap





- Slotted page header contains:
  - Number of record entries
  - End of free space in the block
  - Location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated
- Pointers should not point directly to record — instead they should point to the entry for the record in the header

Organization of records in Files

- **Heap** → A record can be placed anywhere in the file where there is space
- **Sequential** → Store records in sequential order, based on the value of the search key of each record
- **Hashing** → A hash function computed on some attributes of each record; the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file
  - In a ***multitable clustering file organization*** records of several different relations can be stored in the same file
    - Motivation → Store related records on the same block to minimize I/O

Sequential File Organization

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a ***search-key***

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

- Deletion → use pointer chains
- Insertion → Locate the pointer where the record is to be inserted
  - if there is free space, insert there
  - if no free space, insert the record in an ***overflow block***
  - In either case, pointer must chain must be updated
- Need to re-organize the file from time to time to restore sequential order

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

32222	Verdi	Music	48000	

Multitable Clustering File Organization

Store several relations in one file using a ***multitable clustering file organization***

department

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

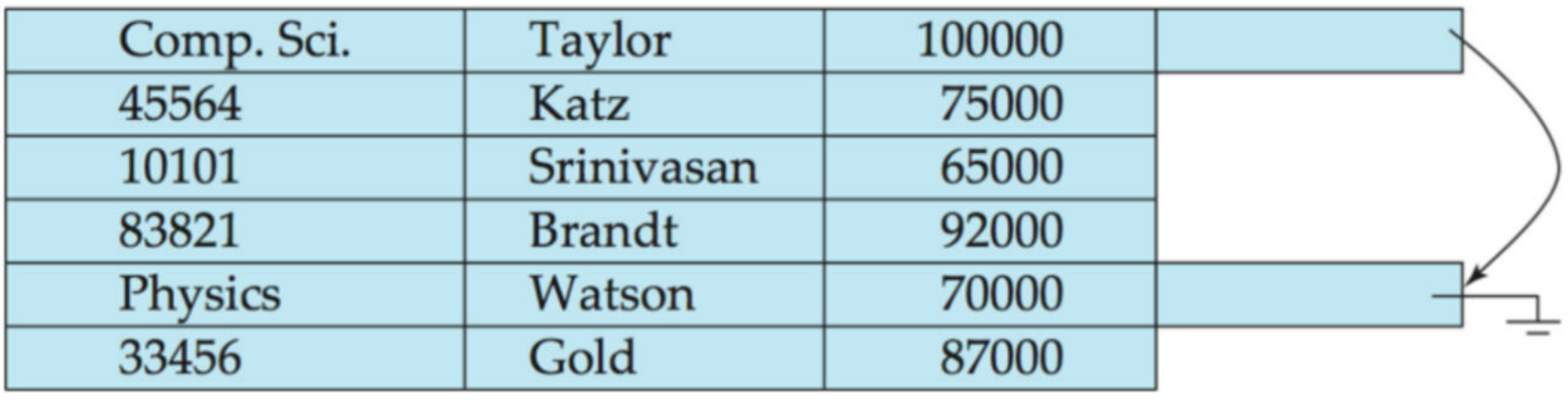
instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

multitable clustering of department and instructor

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

- Good for queries involving $department \bowtie instructor$ and for queries involving one single department and its instructors
- Bad for queries involving only $department$
- Results in variable size records
- Can add pointers chains to link records of a particular relation



Data Dictionary Storage

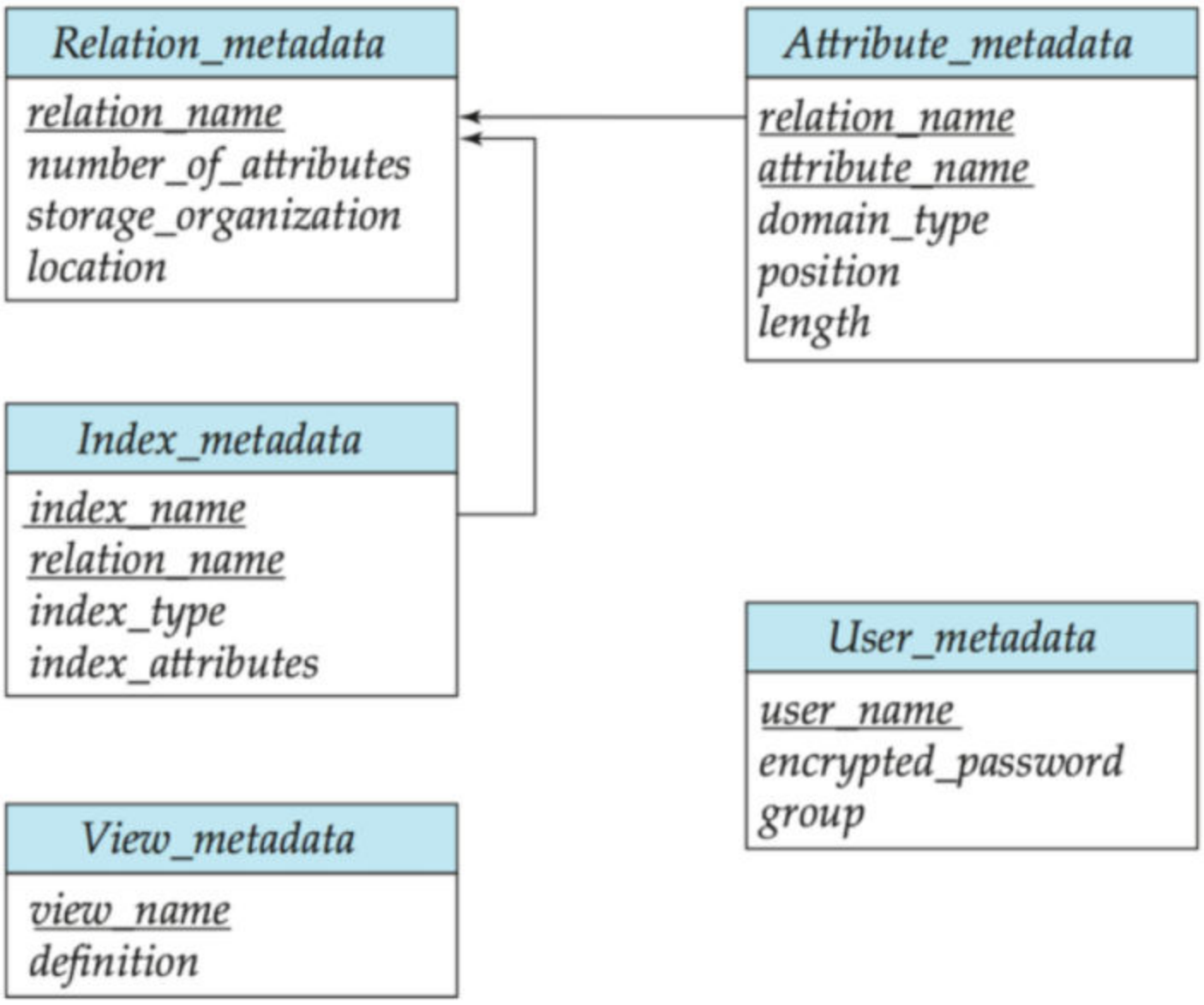
Data Dictionary (also, **System Catalog**) stores ***metadata*** (data about data) such as:

- Information about relations
  - names of relations
  - names, types and lengths of attributes of each relation
  - names and definition of views
  - integrity constraints
- User and accounting information, including password
- Statistical and descriptive data
  - Number of tuples in each relation
- Physical file organization information
  - How relation is stored (sequential/hash/...)
  - Physical location of relation

- Information about indices

Relational Representation of System metadata

- Relational representation on disk
- Specialized data structures designed for efficient access, in memory



Storage Access

- A database file is partitioned into fixed-length storage units called ***blocks***
  - Blocks are units of both storage allocation and data transfer
- Database system seeks to minimize the number of block transfers between the disk and the memory
  - We can reduce the number of disk accesses by keeping as many blocks as possible in the main memory
- **Buffer** → Portion of the main memory available to store copies of disk blocks
- **Buffer Manager** → Subsystem responsible for allocating buffer space in the main memory

Buffer Manager

- Programs call on the buffer manager when they need a block from the disk
  - If the block is already in the buffer, buffer manager returns the address of the block in the main memory
  - If the block is not in the buffer, the buffer manager
    - Allocates space in the buffer of the block
      - Replacing (throwing out) some other block, if required, to make space for the new block
      - Replaced block written back to disk only if it was modified since the most recent time that it was written to / fetched from the disk
    - Reads the block from the disk to the buffer, and returns the address of the block in the main memory to the requester

Buffer Replacement Policies

- Most Operating Systems replace the block ***least recently used (LRU strategy)***
- Idea behind LRU — Use past pattern of block references as a predictor of future references

- Queries have well-defined access patterns (such as sequential scans) and a database system can use the information in a user's query to predict future references
  - LRU may be a bad strategy for certain access patterns involving repeated scans of data
    - For example → When computing the join of 2 relations r and s by a nested loop

      for each tuple tr of r do

        for each tuple ts of s do

          if the tuples tr and ts match ...
  - Mixed strategy with hints on replacement strategy provided by the query optimized is preferable

---

- **Pinned block** → Memory block that is not allowed to be written back to the disk
- **Toss-immediate strategy** → Frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** → System must pin the block currently being processed
  - After the final tuple of that block has been processed, the block is unpinned and it becomes the most recently used block
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
  - For example → the data dictionary is frequently accessed
    - Heuristic → keep data-dictionary blocks in the main memory buffer
- Buffer managers also support forced output of blocks for the purpose of recovery