# Data Structures and Algortihms

Based on lectures by Dr. Arpit Sharma
Notes taken by Rwik Dutta

These notes are not endorsed by the lecturers, and I have modified them (often significantly) after lectures. They are nowhere near accurate representations of what was actually lectured, and in particular, all errors are almost surely mine.[1]

## Contents

[1]This is how Dexter Chua describes his lecture notes from Cambridge. I could not have described mine in any better way.

# 1 Introduction

**Definition 1** (Data Structure)**.** The logical or mathematical model of a particular organization of data.

**Definition 2** (Algorithm)**.** A sequence of computational steps that transform an input into an output. The sequence of steps must terminate for the algorithm to be useful.

## 1.1 Operations on Data Structures

The four major operations that we will focus on in this course are:

1. Traversing: Accessing each record exactly once so that certain items in the record may be processed.
2. Searching: Finding the location of the record with a given key value, or finding the locations of all records which satisfy one or more conditions.
3. Inserting: Adding a new record to the structure.
4. Deleting: Removing a record from the structure.

## 1.2 Asymptotic Notation

**Definition 3** (Asymptotic behaviour)**.** The behaviour of a given function $f(n)$ for very large values of $n$, i.e., as $n \to +\infty$.

In this course, we will work with asymptotically positive functions.

**Definition 4** ($O-$notation)**.** $f(n) = O(g(n))$ if $\exists c, n_0 > 0$ such that $0 \leq f(n) \leq cg(n), \forall n \geq n_0$. Thus, $g(n)$ is an **asymptotic upper bound** of $f(n)$.

**Theorem 1.** Let $f$ be a polynomial of degree $x_0$.

$$f(n) = O(n^x), \forall x \geq x_0$$

**Theorem 2** (Exponential, polynomial, logarithm)**.** Asymptotically, the decreasing order of the functions are: Exponential, polynomial, logarithm.(The bases of the exponential and logarithm functions are greater than 1. The bases will generally by 2 for our purposes.)

**Definition 5** ($\Omega-$notation)**.** $f(n) = \Omega(g(n))$ if $\exists c, n_0 > 0$ such that $0 \leq cg(n) \leq f(n), \forall n \geq n_0$. Thus, $g(n)$ is an **asymptotic lower bound** of $f(n)$.

**Definition 6** ($\Theta-$notation)**.** $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Thus, $g(n)$ bounds $f(n)$ from above and below. Hence, it is an **asymptotically tight bound**.

From now onwards, whenever we compare functions, we will be speaking asymptotically.

## 1.3 Complexity of an Algorithm

The time and space complexities of an algorithm give us an idea of how much time and memory, respectively, is needed by an algorithm to run. In this course, we will only focus on the time complexity. Accordingly, unless otherwise stated or implied, the term "complexity" shall refer to the time complexity of the algorithm.

**Definition 7** (Time complexity)**.** The time complexity of an algorithm is the function $T(n)$ which gives the running time of the algorithm in terms of the size $n$ of the input data. It is often measured by the number of fundamental operations performed.

We have the following types of complexities:

1. Worst case: the maximum $T(n)$ possible.
2. Best case: the minimum $T(n)$ possible.
3. Average case: the expected $T(n)$.

## 2 Recursion

**Algorithm 1** (Factorial)**.** Let $f(n) = n!$ be the factorial function. It can be recursively defined as

$$f(0) = 1$$

$$f(n) = nf(n-1), n > 0$$

factorial($n$):

```
1. If $n = 0$, return 1, exit.
2. Return [$n \times$factorial($n-1$)]
```

**Algorithm 2** (Fibonacci sequence)**.** The Fibonacci sequence is recursively defined as

$$F(0) = F(1) = 1$$

$$F(n) = F(n-1) + F(n-2), n > 1$$

fibo($n$):

```
1. If $n = 0$ or $n = 1$, return 1.
2. Return [fibo($n-1$) + fibo($n-2$)]
```

**Algorithm 3** (Tower of Hanoi)**.** Tower of Hanoi is a mathematical puzzle where we have three rods and $n$ disks of different sizes. Initially, the $n$ disks are placed on one rod in the order of their sizes, from largest to smallest(largest at bottom). The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

Let there be 3 rods A, B, C. Initially all the disks are in A and we want to move everything to C. Also, the disks are numbered from 1 to $n$ in order of their sizes.
TOH($n$, A, C, B):

```
1. If $n = 1$, move disk 1 from A to C.
2. TOH($n-1$, A, B, C)
3. TOH($n-1$, B, C, A)
```

Time Complexity: $O(2^n)$
Recursive equation: $T(n) = 2T(n-1) + 1 \implies T(n) = 2^n - 1$

## 3 Array

**Definition 8** (Linear array)**.** A finite list of $n$ homogenous data elements such that each element is indexed from an index set of $n$ consecutive numbers and the data elements are stored in consecutive memory locations. In this course, we will start indexing arrays from 1.(C and Python start indexing from 0)

### 3.1 Salient Features

- Any element of an array can be accessed in constant time using its index. This is called **random access** to data.
- Arrays are implemented very easily in the memory. However, they are **static**.
- Most sorting techniques can be used in arrays very efficiently due to the random access. **Randomized quicksort** is used in most practical cases($O(n \log n)$).
- Searching in a sorted array is faster compared to other data structures. Binary search is the preferred search algorithm($O(\log n)$).
- Insertion and deletion in arrays are not very efficient.

## 3.2 Searching

### 3.2.1 Linear Search

**Algorithm 4** (Linear search)**.** Let ARR be a linear array containing $n$ elements. We are searching for VALUE.

1. Set $i = 1$.
2. If ARR[$i$] = VALUE, exit. We have found VALUE at index $i$.
3. If $i = n$, exit. VALUE not found in ARR.
4. Increment $i = i + 1$. Go to 2.

Time complexity:
Worst case: $\Theta(n)$

From now onwards, the division of two integers will represent the following, unless specified otherwise: $\frac{p}{q} := \lfloor \frac{p}{q} \rfloor$ or $\lceil \frac{p}{q} \rceil$, whichever is convenient.

### 3.2.2 Binary Search

Unlike, linear search

- the list must be sorted
- one must have direct access to the middle element in any sublist

**Algorithm 5** (Binary search)**.** Let ARR be a sorted(ascending order) linear array containing $n$ elements. We are searching for VALUE.

1. Set $l = 1, r = n$.
2. (Go to the middle element)Set $i = \frac{l+r}{2}$.
3. If ARR[$i$] = VALUE, exit. We have found VALUE at index $i$.
4. If $l \geq r$, exit. VALUE not found in ARR.
5. If ARR[$i$] < VALUE, (search right subarray)set $l = i + 1$. Go to 2.
6. If ARR[$i$] > VALUE, (search left subarray) set $r = i - 1$. Go to 2.

Time complexity:
Worst case: $\Theta(\log n)$

## 3.3 Insertion

**Algorithm 6** (Insertion)**.** To insert in a linear array at a given position, we shift every element to the right of that position, one place to the right.

1. Set $i = n$.
2. If $i <$ INDEX, ARR[INDEX] = VALUE, set $n = n + 1$, exit.
3. ARR[$i + 1$] = ARR[$i$]
4. $i = i - 1$. Go to 2.

Time complexity:
Best case: $O(1)$, insert at end.
Worst case: $O(n)$, insert at begining.

### 3.4 Deletion

**Algorithm 7** (Deletion)**.** Deletion is similar to insertion. We move all elements to the right of the index, one place to the left. A copy of the last element remains after the end of the new array.

1. Set $i$ = INDEX.
2. If $i = n$, set $n = n - 1$, exit.
3. ARR[$i$] = ARR[$i + 1$].
4. Set $i = i + 1$. Go to 2.

Time complexity:
Best case: $O(1)$, delete at end.
Worst case: $O(n)$, delete at begining.

## 4 Linked List

**Definition 9** (Linked list)**.** A linked list is a linear data structure, in which the elements are not stored at continuous memory locations. It consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

The starting node is called the HEAD of the linked list. Here is a singly-linked list.

$$\boxed{\text{HEAD}} \to \boxed{\text{NODE 1}} \to \boxed{\text{NODE 2}} \to \boxed{\text{NODE 2}} \to \boxed{\text{NULL}}$$

Here is a doubly-linked list.

$$\boxed{\text{HEAD}} \to \boxed{\text{NODE 1}} \leftrightarrow \boxed{\text{NODE 2}} \leftrightarrow \boxed{\text{NODE 3}} \leftarrow \boxed{\text{TAIL}}$$

### 4.1 Salient Features

- There is no random access to each node. For traversal, one must start at the HEAD and go through each node.
- Insertion and deletion are simple and fast in linked lists compared to arrays.
- Binary search cannot be used. Thus, searching is slower than arrays.
- These are **dynamic** data structures.
- Sorting techniques that require a lot of random access are slower in linked lists(like, quicksort). However, we can use selection sort, bubble sort and insertion sort(for doubly-linked lists). **Merge sort** is preferred for linked list due to its superior complexity.

### 4.2 Insertion

**Algorithm 8** (Insertion at the beginning)**.** This can be done in $O(1)$ time. For a doubly-linked list, we can similarly insert at the end.

1. Set TEMP = HEAD
2. Set HEAD = new_NODE
3. Set new_NODE → NEXT = TEMP, exit.

**Algorithm 9** (Insertion at other positions)**.** We first need to reach the node before the insertion point to do the insertion.

1. Set PTR = HEAD. Set $i = 2$.
2. If $i$ = INDEX, set new_NODE → NEXT = PTR → NEXT, set PTR → NEXT = new_NODE, exit.
3. PTR = PTR → NEXT. $i = i + 1$. Go to 2.

Time Complexity:
Worst case: $O(n)$

**Algorithm 10** (Insertion at the end)**.**

1. Set PTR = HEAD. Set new_NODE → NEXT = NULL.
2. If PTR → NEXT = NULL, set PTR → NEXT = new_NODE, exit.
3. PTR = PTR → NEXT. Go to 2.

## 4.3 Deletion

**Algorithm 11** (Delete at the beginning)**.** Similar to insertion.

```
1. Set TEMP = HEAD.
2. Set HEAD → NEXT = HEAD
3. Delete TEMP. Exit.
```

**Algorithm 12** (Deletion at other positions)**.** For deletion in singly-linked list, we must reach the node before the node to be deleted.

```
1. Set PTR = HEAD. Set $i = 2$.
2. If $i$ = INDEX, set TEMP = PTR → NEXT, set PTR → NEXT = PTR → NEXT → NEXT, delete TEMP, exit.
3. PTR = PTR → NEXT. $i = i + 1$. Go to 2.
```

**Algorithm 13** (Deletion at the end)**.**

```
1. Set PTR = HEAD. Set new_NODE → NEXT = NULL.
2. If PTR → NEXT → NEXT = NULL, set TEMP = PTR → NEXT, set PTR → NEXT = NULL, delete TEMP, exit.
3. PTR = PTR → NEXT. Go to 2.
```

# 5 Sorting Linear Data

## 5.1 Selection Sort

**Algorithm 14** (Selection sort)**.** There are two subarrays maintained: sorted(left), unsorted(right). In every iteration(called **pass**), the minimum element from the unsorted subarray is picked and moved to the sorted subarray by swapping it with the first element of the unsorted subarray.

```
1. Set $i = 1$.
2. If $i = n$, exit. ARR is sorted.
3. Set $min = i$. Set $j = i + 1$.
4. If $j \leq n$
    (a) If ARR[$j$] < ARR[$min$], set $min = j$.
    (b) Increment $j = j + 1$. Go to 4.
5. swap(ARR[$min$], ARR[$i$])
6. Increment $i = i + 1$. Go to 2.
```

Time complexity:
 Worst case: $O(n^2)$, array is reverse sorted. $O(n)$ swaps.
 Best case: $O(n^2)$, array is already sorted. $O(1)$ swaps.
 Average case: $O(n^2)$. $O(n)$ swaps.
Advantages:

1. Efficient use in linked lists.

Disadvantage:

1. Even in the best case, complexity is $O(n^2)$.

**Example 1.** Let ARR $= [64, 25, 12, 22, 11]$. The passes of selection sort are as follows:

$$\underbrace{11}_{sorted} \mid \underbrace{25, 12, 22, 64}_{unsorted}$$

$$11, 12|25, 22, 64$$

$$11, 12, 22|25, 64$$

$$11, 12, 22, 25|64 \qquad \text{(sorted)}$$

## 5.2 Insertion Sort

**Algorithm 15** (Insertion sort)**.** The insertion sort uses the process of insertion in arrays. In a pass, an element is picked from its original position and inserted in way that all elements to its left are less than it.

```
1. Set $i = 2$.
2. If $i > n$, exit. ARR is now sorted.
3. Set $k =$ ARR[$i$]. Set $j = i - 1$.
4. If $j >= 1$ and ARR[$j$] > k,
    (a) ARR[$j + 1$] = ARR[$j$]
    (b) Decrement $j = j - 1$. Go to 4.
5. Set ARR[$j + 1$] = $k$.
6. Increment $i = i + 1$. Go to 2.
```

Time complexity:
  Worst case: $O(n^2)$, array is reverse sorted.
  Best case: $O(n)$, array is sorted.
  Average case: $O(n^2)$
Advantages:

1. Adaptive: total number of steps is reduced for partially sorted array.
2. Less swaps than bubble sort.

**Example 2.** Let ARR $= [12, 11, 13, 5, 6]$. The passes of insertion sort are as follows:

$$12, \boxed{11}, 13, 5, 6$$

$$11, 12, \boxed{13}, 5, 6$$

$$11, 12, 13, \boxed{5}, 6$$

$$5, 11, 12, 13, \boxed{6}$$

$$5, 6, 11, 12, 13$$

## 5.3 Bubble Sort

**Algorithm 16** (Bubble sort)**.** Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. After each pass, the greatest element in the subarray reaches the correct position.

```
1. Set $i = 1$.
2. If $i = n$, exit. ARR is sorted.
3. Set $j = 1$.
4. If $j \leq (n - i)$
    (a) If ARR[$j$] > ARR[$j + 1$], swap(ARR[$j$],ARR[$j + 1$]).
    (b) Increment $j = j + 1$.
    (c) Go to 4.
5. Increment $i = i + 1$. Go to 2.
```

Time complexity:
  Worst case: $O(n^2)$, array is reverse sorted. $O(n^2)$ swaps.
  Best case: $O(n^2)$, array is already sorted. This algorithm can be optimized by stopping the algorithm if inner loop didn't cause any swap(identifying that the array is already sorted). In this case, we get $O(n)$ with $O(1)$ swaps.
  Average case: $O(n^2)$. $O(n^2)$ swaps.
Advantages:

1. Stable: does not change the relative order of elements with equal keys.
2. Using optimized approach, it can detect already sorted array in first pass with time complexity of $O(1)$.

**Example 3.** Let ARR $= [5, 1, 4, 2, 8]$.
Pass 1:

$$\boxed{5}, 1, 4, 2, 8$$

$$1, \boxed{5}, 4, 2, 8$$

$$1, 4, \boxed{5}, 2, 8$$

$$1, 4, 2, \boxed{5} 8$$

$$1, 4, 2, 5|8$$

Pass 2:

$$\boxed{1}, 4, 2, 5|8$$

$$1, \boxed{4}, 2, 5|8$$

$$1, 2, \boxed{4}, 5|8$$

$$1, 2, 4|5, 8$$

Pass 3:

$$\boxed{1}, 2, 4|5, 8$$

$$1, \boxed{2}, 4|5, 8$$

$$1, 2|4, 5, 8$$

Pass 4:

$$\boxed{1}, 2|4, 5, 8$$

$$1|2, 4, 5, 8$$

In the optimized algorithm, this ends at pass 3.

## 5.4 Merge Sort

**Algorithm 17** (Sorted merge)**.** Sorted merge is used to merge two sorted arrays into a sorted array. There are two pointers $P, Q$ that move through the elements of ARR1 and ARR2, respectively. The smaller of the values at P and Q is appended to the resultant array and that pointer moves to the right.

```
1. Set $p = q = 1$.
2. If $p > n_1$, set ARR1$[p] = +\infty$.
3. If $q > n_2$, set ARR2$[q] = +\infty$.
4. If $p = q = +\infty$, exit.
5. If ARR1$[p] <$ ARR2$[q]$,
    (a) RESULT.append(ARR1$[p]$)
    (b) Increment $p = p + 1$. Go to 2.
6. If ARR2$[q] <$ ARR1$[p]$,
    (a) RESULT.append(ARR2$[q]$)
    (b) Increment $q = q + 1$. Go to 2.
```

Time Complexity:
Worst case: $O(n_1 + n_2)$

**Example 4.** Let us merge ARR1 $= [1, 5, 8, 15, 24]$ and ARR2 $= [3, 10, 16, 20]$ into a sorted array.

| ARR1 | ARR2 | Result |
|---|---|---|
| $[\boxed{1}, 5, 8, 15, 24]$ | $[\boxed{3}, 10, 16, 20]$ | $[\ ]$ |
| $[1, \boxed{5}, 8, 15, 24]$ | $[\boxed{3}, 10, 16, 20]$ | $[1]$ |
| $[1, \boxed{5}, 8, 15, 24]$ | $[3, \boxed{10}, 16, 20]$ | $[1, 3]$ |
| $[1, 5, \boxed{8}, 15, 24]$ | $[3, \boxed{10}, 16, 20]$ | $[1, 3, 5]$ |
| $[1, 5, 8, \boxed{15}, 24]$ | $[3, \boxed{10}, 16, 20]$ | $[1, 3, 5, 8]$ |
| $[1, 5, 8, \boxed{15}, 24]$ | $[3, 10, \boxed{16}, 20]$ | $[1, 3, 5, 8, 10]$ |
| $[1, 5, 8, 15, \boxed{24}]$ | $[3, 10, \boxed{16}, 20]$ | $[1, 3, 5, 8, 10, 15]$ |
| $[1, 5, 8, 15, \boxed{24}]$ | $[3, 10, 16, \boxed{20}]$ | $[1, 3, 5, 8, 10, 15, 16]$ |
| $[1, 5, 8, 15, \boxed{24}]$ | $[3, 10, 16, 20]$ | $[1, 3, 5, 8, 10, 15, 16, 20]$ |
| $[1, 5, 8, 15, 24]$ | $[3, 10, 16, 20]$ | $[1, 3, 5, 8, 10, 15, 16, 20, 24]$ |

**Algorithm 18** (Merge sort)**.** This is a **divide and conquer algorithm**. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

```
mergeSort(ARR[$l \to r$]):
  1. If $l \geq r$, exit.
  2. Set $m = \frac{l+r}{2}$.
  3. mergeSort(ARR[$l \to m$])
  4. mergeSort(ARR[$m + 1 \to r$])
  5. sortedMerge(ARR[$l \to m$],ARR[$m + 1 \to r$])
```

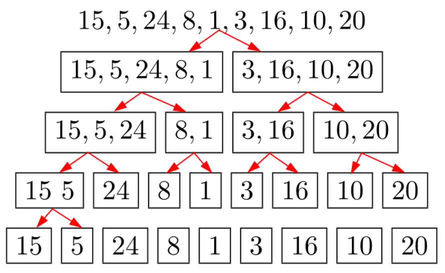
Time Complexity:
All cases: $O(n \log n)$
Advantages:

1. Merge sort can be used for linked list as data is accessed sequentially.

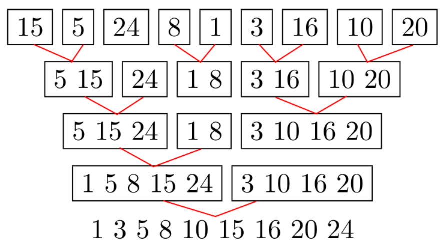
Disadvantage:

1. Goes through the entire process even if the array is sorted.

**Example 5.** Divide:($O(1)$)



Merge:($O(n)$)



## 5.5 Quick Sort

**Algorithm 19** (Quick sort)**.** Like merge sort, quick sort is a divide and conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways. It is seen that randomized quicksort works very well in practical cases.

After each pass, all elements to the left of the pivot are smaller than the pivot and all elements to the right of the pivot are larger than the pivot, thus the pivot is in the correct position.

1. Select the pivot.
2. Use the partition method to get the pivot to its correct position.
3. Repeat this for the subarrays formed to the left and right of the pivot.

Time complexities:
  Worst case: $O(n^2)$
  Best and average case: $O(n \log n)$. QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

**Algorithm 20.** (Partition) We will now see how the partition method in quicksort works. Let our pivot be the first element.

1. Set $p = 2, q = n, \text{pivot} = 1$. Set ARR$[0] = infty$, ARR$[n+1] = +\infty$.
2. Until ARR$[p] >$ ARR[pivot], $p = p + 1$.
3. Until ARR$[q] <$ ARR[pivot], $q = q - 1$.
4. If $p < q$, swap(ARR$[p]$,ARR$[q]$). Go to 2.
5. If $p \geq q$, swap(ARR[pivot],ARR$[q]$). Exit.

**Example 6.** Let ARR $=[35, 50, 15, 25, 80, 20, 90, 45]$.

$$\boxed{35}, \underbrace{50}_{P}, 15, 25, 80, 20, 90, \underbrace{45}_{Q}$$

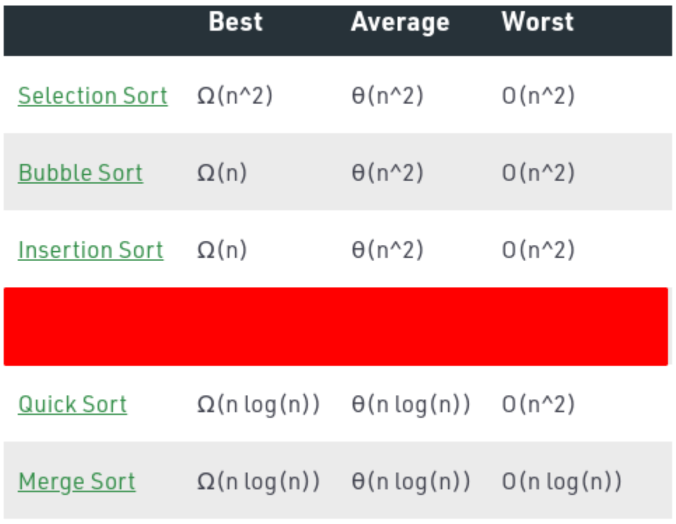
$$\boxed{35}, \underbrace{50}_{P}, 15, 25, 80, \underbrace{20}_{Q}, 90, 45$$

$$\boxed{35}, \underbrace{20}_{P}, 15, 25, 80, \underbrace{50}_{Q}, 90, 45$$

$$\boxed{35}, 20, 15, \underbrace{25}_{Q}, \underbrace{80}_{P}, 50, 90, 45$$

$$25, 20, 15, \boxed{35}, 80, 50, 90, 45$$

## 5.6 Comparison of sorting algorithms



| | Best | Average | Worst |
|---|---|---|---|
| Selection Sort | Ω(n^2) | θ(n^2) | O(n^2) |
| Bubble Sort | Ω(n) | θ(n^2) | O(n^2) |
| Insertion Sort | Ω(n) | θ(n^2) | O(n^2) |
| [redacted] | [redacted] | [redacted] | [redacted] |
| Quick Sort | Ω(n log(n)) | θ(n log(n)) | O(n^2) |
| Merge Sort | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |

## 6 Stack

Stacks are a kind of data structure where we are only interested in the last inserted element at any point of time. It is a LIFO(Last In First Out) data structure. The last inserted element is called the TOP. A stack can be implemented by an array or a linked list.

$$1\ 9\ 3\ 8\ 5 \leftarrow \boxed{\text{top}}$$

## 6.1 Push

**Algorithm 21** (Push)**.** Push is a stack operation that inserts an element to the top of the stack.

1. Set TOP = TOP + 1
2. Set STACK[TOP] = VALUE. Exit.

## 6.2 Pop

**Algorithm 22** (Pop)**.** The pop operation removes the top element from the stack and returns its value.

1. Set VALUE = STACK[TOP]
2. Set TOP = TOP - 1.
3. Return VALUE. Exit.

There is an operation similar to POP where we only return the value of the TOP element but do not change the TOP pointer. It is called **PEEK**.

All operations take $O(1)$ time(even for linked list implementation).

## 6.3 Evaluation of expressions

Generally, we write an arithmetic expression in its **infix** form where the operator lies between the operands. However, such an expresstion cannot be computed from left to right(or, right to left) due to the precedence of operators. Thus, we convert it into **postfix** form for such computations. A stack is used for this purpose.

**Algorithm 23** (Infix to Postfix)**.** Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push '(' onto STACK, and add ')' to the end of Q. Scan Q from left to right and repeat the following steps.
2. If STACK is empty, exit. P is the required postfix expression.
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator $\circ$ is encountered,
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than $\circ$.
   (b) Add $\circ$ to STACK.
6. If a right parenthesis is encountered,
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   (b) Remove the left parenthesis from the STACK.

**Algorithm 24** (Evaluation of postfix expression)**.** Now that we have converted our infix to postfix, we would like to evaluate the expression.

1. Add a right parenthesis ) at the end of P. Scan P from left to rightand repeat the following steps.
2. If only a ) is left to be scanned in P, exit. VALUE = STACK[TOP].
3. If an operand is encountered, put it on STACK.
4. If an operator $\circ$ is encountered,
   (a) Pop the two top elements of STACK, where A is the top element and B is the next-to-top element.
   (b) Evaluate B $\circ$ A and push the result into the stack.

# 7 Queue

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).

$$\boxed{\text{front}} \rightarrow 1\ 9\ 3\ 8\ 5 \leftarrow \boxed{\text{rear}}$$

## 7.1 Salient Features

- Both insertion and deletion is done in $O(1)$ time.
- Array implementation is often problematic for large number of enque and deque operations. So, we may use linked list(time complexity remains same).
- We often use **circular queues** for better utilisation of space. Also, we only need to have a pointer for the front/rear as the other is always next to it.
- We can **implement a queue using stack**. We will require two stacks. We can also **implement a stack using queue**. We will need two queues.

## 7.2 Enque

**Algorithm 25** (Enque)**.** Insert element to the REAR.

```
1. Set REAR = REAR + 1
2. Set QUEUE[REAR] = VALUE. Exit.
```

## 7.3 Deque

**Algorithm 26** (Deque)**.** Remove element from the FRONT and return its value.

```
1. Set VALUE = QUEUE[FRONT]
2. Set FRONT = FRONT + 1. Exit.
```

## 7.4 Priority Queue

**Definition 10** (Priority queue)**.** Priority Queue is an extension of queue with following properties.

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

It is often implemented using heaps.

# 8 Binary Tree

Until now, we have only studied **linear data structures**. Trees are **hierarchical data structures**.

**Definition 11** (Root, children, parent, sibling, leaf, edge)**.** The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. Two nodes with the same parent are called siblings. A node that has no children is called a leaf node. All other nodes(except root) are called edges.

**Definition 12** (Level or depth)**.** The number of edges on the path from the root to a node is called its level or depth. The root, unless mentioned otherwise, is at level 1.

**Definition 13** (Height)**.** Height of a tree is the maximum level for any node in the tree.

**Definition 14** (Binary tree)**.** A tree where each node has at most 2 children. These are called the left child and the right child.

## 8.1 Salient features

- The maximum number of nodes at level $l$ of a binary tree is $2^l$.
- The maximum number of nodes in a binary tree of height $h$ is $2^h - 1$.
- In a binary tree with $n$ nodes, minimum possible height is $\log_2(n+1)$.
- A binary tree with $L$ leaves has at least $\lceil \log_2 L \rceil + 1$ levels.
- In binary tree where every node has 0 or 2 children, the number of leaf nodes is always one more than nodes with two children.
- Insertion and deletion has no rules except the maximum of 2 children. This is can be done really fast. This is faster than arrays but slower than linked lists.
- Searching items in a general binary tree is slow. As there is no particular order, every node may have to be checked in the worst case.

## 8.2 Types of binary trees

**Definition 15** (Full binary tree)**.** Every node has 0 or 2 children.

**Definition 16** (Complete binary tree)**.** All the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

**Definition 17** (Perfect binary tree)**.** All the internal nodes have two children and all leaf nodes are at the same level.

**Definition 18** (Balanced binary tree)**.** $h = O(\log n)$

**Definition 19** (Degenrate binary tree)**.** Every internal node has one child.

## 8.3 Traversal

Unlike linear data structures which have only one logical way to traverse them, trees can be traversed in different ways.

- Inorder: Left, Root, Right
- Preorder: Root, Left, Right
- Postorder: Left, Right, Root
- Level order: Each level, left to right

All traversals can be done in $O(n)$ time. The first three are called depth-first traversals. The inorder traversal along with any other depth-first traversal can be used to recontruct a binary tree.

## 8.4 Threading

A binary tree is made threaded by making all right child pointers that would normally be NULL, point to the inorder successor of the node (if it exists). The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion.

# 9 Binary Search Tree

As the name suggests, a binary search tree is a form of binary tree that makes searching easier. We lose some time in insertion and deletion.

**Definition 20** (Binary search tree)**.** A binary tree with following rules:

1. The left subtree of a node contains only nodes with keys lesser than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtree each must also be a binary search tree.

Searching, insertion and deletion all have a complexity of $O(h)$. In the worst case, this is $O(n)$.

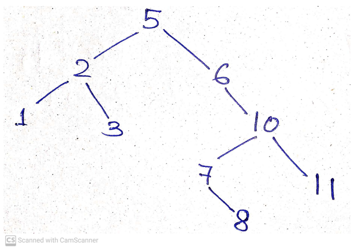
## 9.1 Searching

Searching in BST is faster than unordered linked lists but slower than arrays.

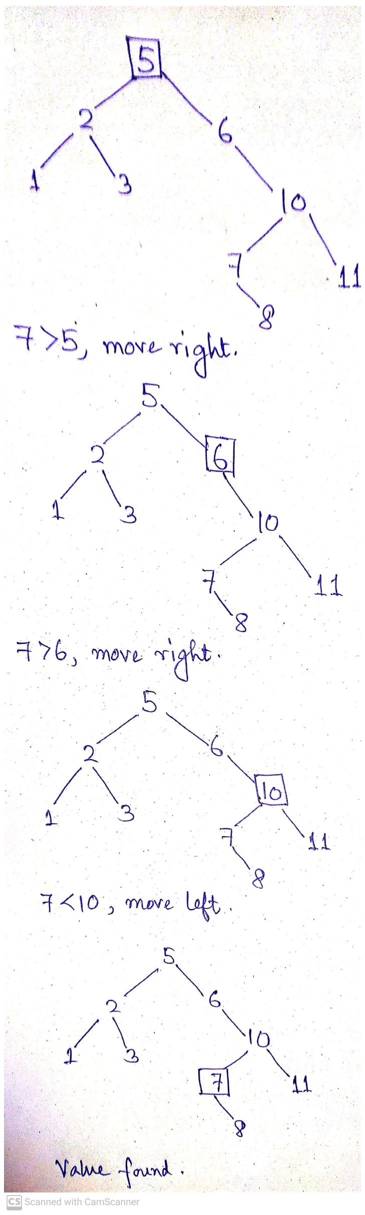
**Algorithm 27.** bstSearch(tree, VALUE):

```
1. Set NODE = ROOT.
2. If NODE is NULL, exit. VALUE not found.
3. If NODE.key = VALUE, VALUE found at NODE. Exit.
4. If VALUE < NODE.key, bstSearch(left subtree, VALUE).
5. If VALUE > NODE.key, bstSearch(right subtree, VALUE).
```

**Example 7.** Let the BST be:



The search of the key 7 happens like:

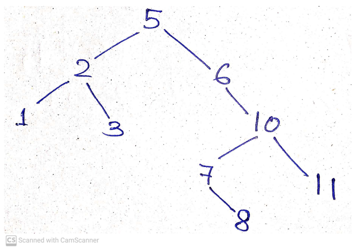


## 9.2 Insertion

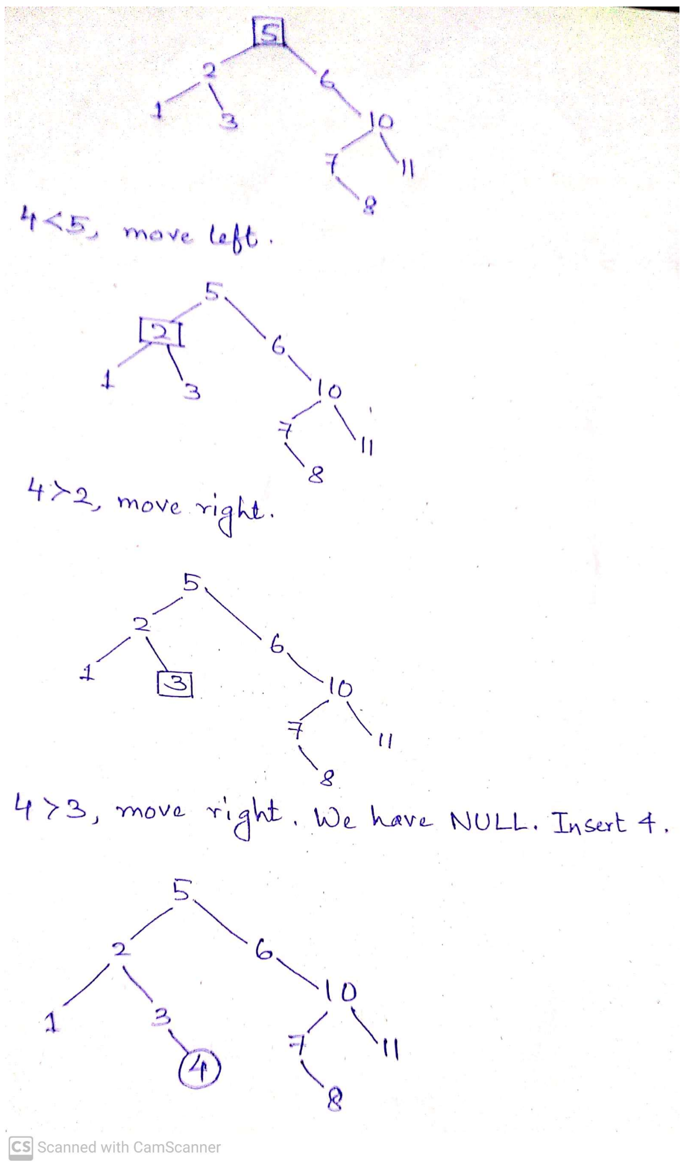
**Algorithm 28.**
insert(tree, VALUE)

```
1. Set NODE = ROOT.
2. If NODE = NULL, set NODE.key = VALUE. Set NULL as NODE's as its children. Exit.
3. If VALUE < NODE.KEY, insert(left subtree, VALUE)
4. If VALUE > NODE.KEY, insert(right subtree, VALUE)
```

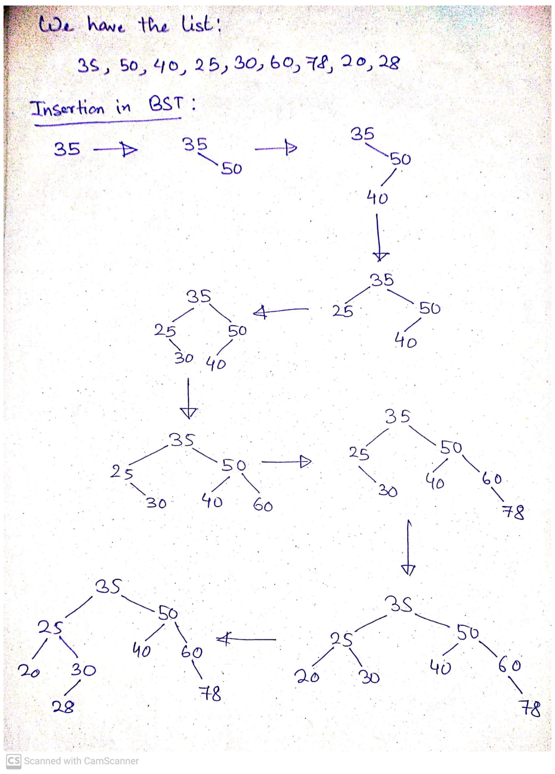
**Example 8.** Let the BST be:



We need to insert 4.



**Example 9.** We have to insert a given list of numbers in order.



## 9.3 Deletion

**Algorithm 29.**

- If node to be deleted is the leaf: Simply remove from the tree.
- If node to be deleted has only one child: Copy the child to the node and delete the child.
- If node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

## 9.4 AVL Search Tree

As we have seen, all the algorithms require $O(h)$ time. Thus, we would like to reduce the height of a tree as much as possible.

**Definition 21** (Balance factor)**.** The balance factor of a node is given by

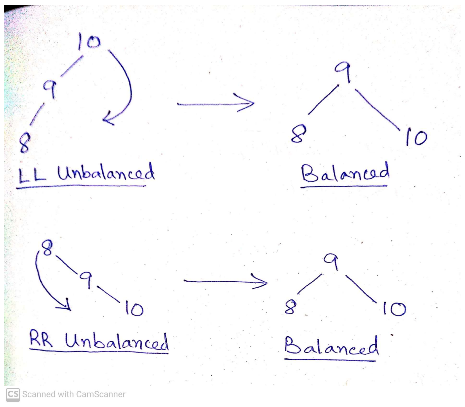
$$\text{Balance factor} = h(\text{left subtree}) - h(\text{right subtree})$$

**Definition 22** (AVL search tree)**.** A balanced binary search tree(i.e., $h = O(\log n)$) such that the balance factor of each node is in the set $\{-1, 0, 1\}$.

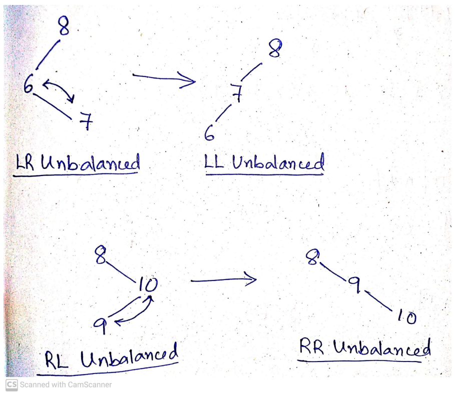
AVL search trees can be used when we need to do a lot of searching but insertion and deletion are not done frequently.

### 9.4.1 Balancing

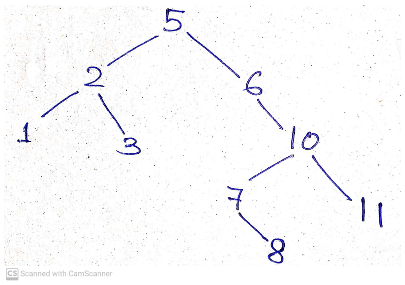
We have 4 types of unbalancedness in a BST.
LL and RR only require 1 rotation to balance.



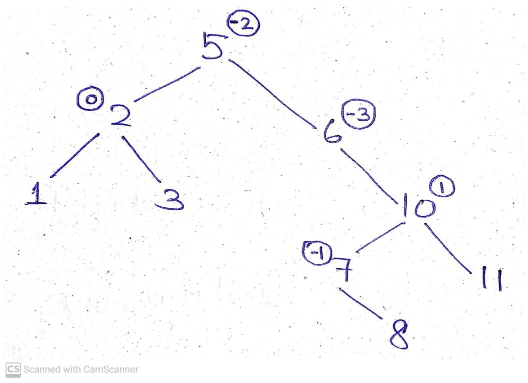
LR and RL first need to be converted to LL and RR, respectively. Thus, they require 2 rotations to balance.



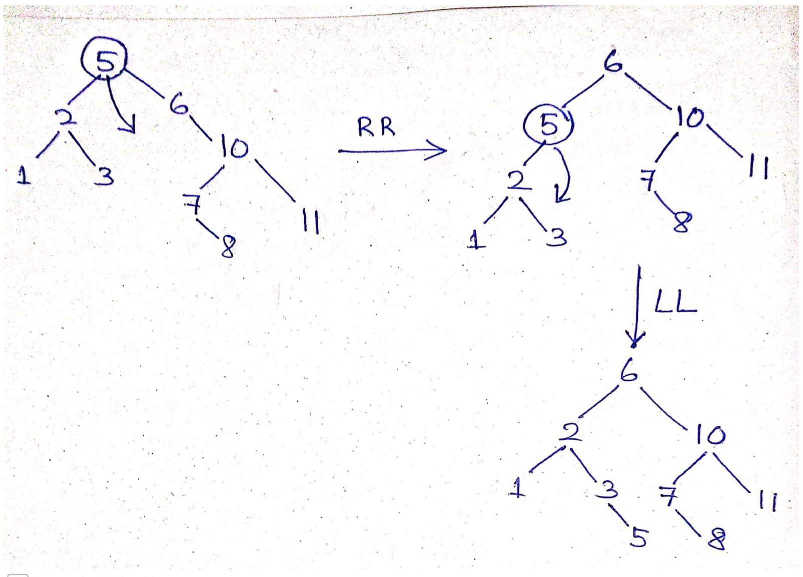
So, how do we balance



We note down the balance factor of each node. Leaf nodes always have a balnce factor of 0.



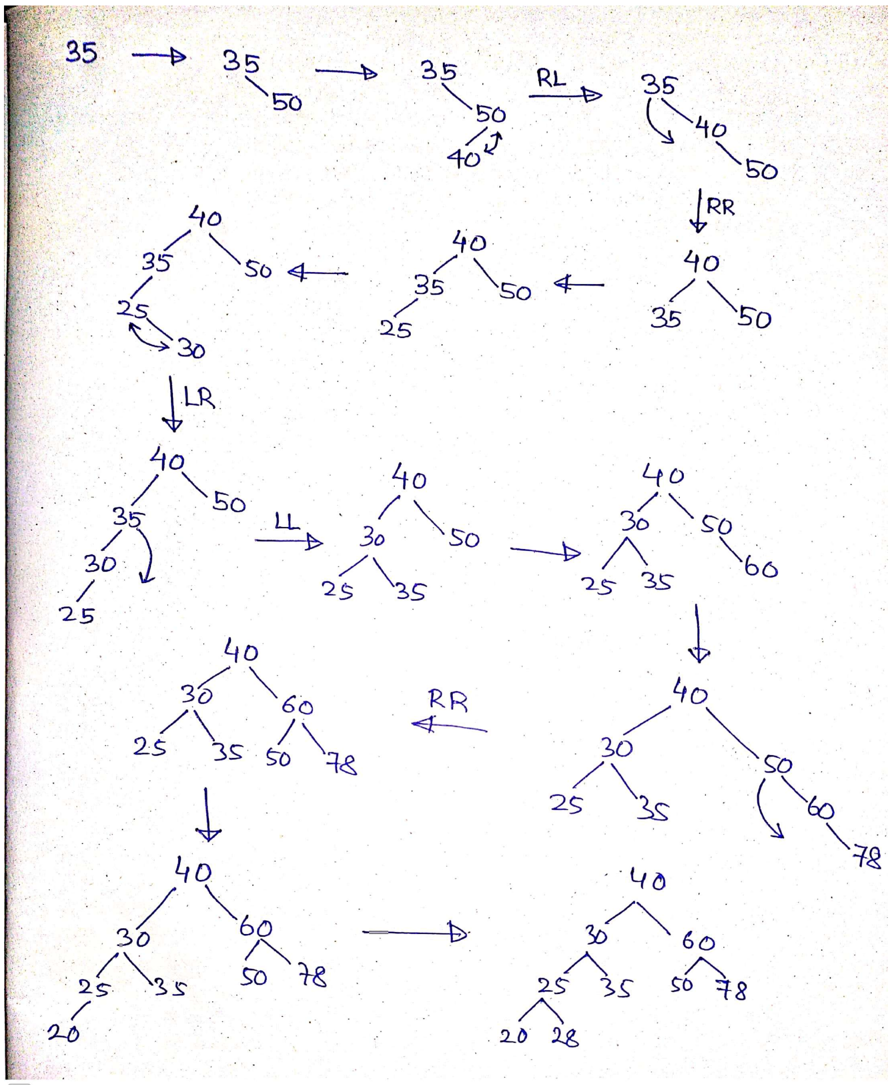
Clearly the BST is unbalanced. We want to convert it to an AVL search tree.
We perform the following rotations.



### 9.4.2 Insertion

We insert just like a general BST. However, we keep track of the balance factor of the nodes and balance the tree whenever we encounter some unbalncedness(bottome to top). At the end of an insertion, the tree must be balanced.

**Example 10.** Suppose we need to insert the following list in order: 35, 50, 40, 25, 30, 60, 78, 20, 28.



We required 4 rotations.

As we can see, all operations take $O(\log n)$ time now.

## 9.5 Red Black Tree

A red black tree is also a balnced BST. It is somewhat less balanced than AVL search tree, so, search is slower. Red black trees are preferred when we need to do frequent insertions and deletions.

**Definition 23** (Red black tree)**.** A balanced BST with the following features:

- Every node has a colour either red or black.
- The root of the tree is always black.
- Every NULL leaf is black.
- There are no two adjacent red nodes (A red node cannot have a red parent or red child).
- Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.

Every RBT with $n$ nodes has $h \leq 2\log_2(n+1)$.

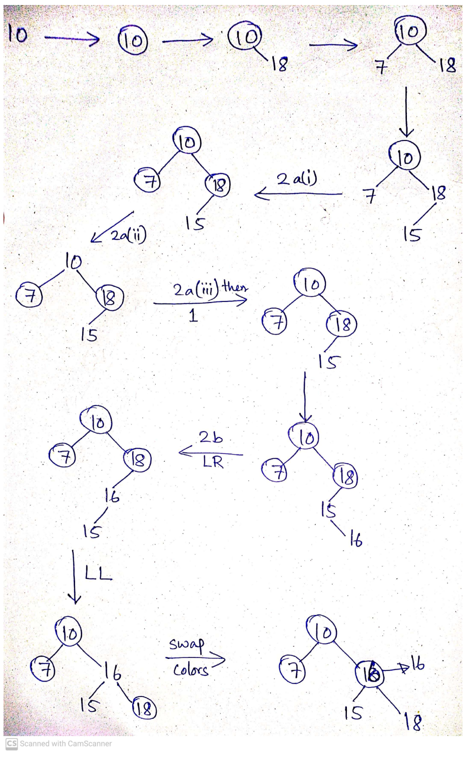
### 9.5.1 Insertion

While AVL trees always require rotations to balance, RBTs may require just recoloring of nodes.

**Algorithm 30.** Let X be the newly inserted node. Perform standard BST insertion and make the colour of newly inserted nodes as RED.

1. If X is the root, change the colour of X as BLACK.
2. If X's parent RED and X is not the root,
   (a) If X's uncle is RED (Grandparent must have been black from property 4).
      i. Change the colour of parent and uncle as BLACK.
      ii. Colour of a grandparent as RED.
      iii. Set X = X's grandparent, Go to 1.
   (b) If X's uncle is BLACK, do required rotation. After LL or RR rotation, swap the colours of the rotated node and the node about which rotation occured.

**Example 11.** Suppose, we insert the numbers 10, 18, 7, 15, 16 to an empty tree. Circled nodes are black.



# 10 m-way Search Tree

**Definition 24** ($m-$way search tree)**.** The m-way search trees are multi-way trees which are generalised versions of binary search trees where each node contains multiple elements. Inside a node, elements are in order. The subtree between two keys $k_1$ and $k_2$ contains only keys in the range from $k_2$ to $k_2$.

**Definition 25** (Order)**.** The order denotes the maximum number of children a node can have. If the order is $t$, a node can have at most $t-1$ elements.

- The number of elements in an $m$-way search tree of height $h$ ranges from a minimum of $h$ to a maximum of $m^h - 1$.
- An $m$-way search tree of $n$ elements ranges from a minimum height of $log_m(n+1)$ to a maximum of $n$.
- Operations are similar to BST and require $O(h)$ time.

## 10.1 B tree

**Definition 26** (B tree)**.** A self-balncing $m-$way search tree with the properties:($t$ is the order)

- All leaves are at the same level.
- Every node except root must contain at least $\lceil \frac{t}{2} \rceil - 1$ keys. The root may contain minimum 1 key.
- Number of children of a node is equal to the number of keys in it plus 1.

Minimum height: $\lceil \log_m(n+1) \rceil - 1$
Maximum height: $\lceil \log_t \frac{n+1}{2} \rceil$
Complexity for all operations is $O(\log n)$

The drawback of B tree used for **dynamic multilevel indexing** is that it stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B tree. This technique, greatly reduces the number of entries that can be packed into a node of a B tree, thereby contributing to the increase in the number of levels in the B tree, hence increasing the search time of a record.

## 10.2 B+ tree

A B+ tree with $l$ levels can store more entries in its internal nodes compared to a B tree having the same $l$ levels.

- The structure of leaf nodes is quite different from the structure of internal nodes.
- Data pointers present only at the leaf nodes of the tree.
- Leaf nodes store all the key values along with their corresponding data pointers.
- The leaf nodes are linked to provide ordered access to the records.
- Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.
- A B+ tree, unlike a B tree has two orders, $a$ and $b$, one for the internal nodes and the other for the external (or leaf) nodes.