Lecture Notes – COMP2119

Notes by José A. Espiño P. *

Semester 2 2022–2023

Contents

*The content in this notes is sourced from what was covered during lectures, in the book *Introduction to Algorithms* by Thomas H. Cormen et al, or in websites such as Geeksforgeeks.org and Programiz.com. I claim no autorship over the contents herein.

```
Sample!
```

1 Induction, Recurrence Equations, and Recursive Calls

A mathematical function is recursive if it is defined in terms of itself, for example:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n-1) + n^2 & \text{if } n > 0 \end{cases}$$

The above equation is called a recurrence equation.
The value of a certain recursive function given certain parameter value(s) can be obtained by **telescoping** until a **base case** is reached. For example, $f(3)$ can be computed as:

$$\begin{aligned} f(3) &= f(2) + 3^2 \\ &= f(1) + 2^2 + 9 \\ &= f(0) + 1^2 + 13 \\ &= 14 \end{aligned}$$

There are two important rules when it comes to defining a recursive function:

1. A base case, which is the case for which the value of the function is directly known without resorting to recursion. There can be several base cases.

2. Progress. The recursive call must always make progress towards a base case and grow smaller.

In a computer program, we can have recursive functions or procedures. These should also follow the aforementioned rules of recursion.
Take a look at this example:

```
int f(n){  /* assume n>=0*/
  if(n==0) return (0);
  else return(f(n-1)+n*n);
}
```

Recursive programming utilizes the solution of a smaller problem to solve a larger one.
Often, we can write recursive problems in an iterative manner. These might be more or less efficient than their recursive counterpart — it depends on the problem at hand. However, if the two algorithms were to perform the same

amount of operations, we generally prefer the iterative approach because recursion requires more computer resources.

Mathematical Induction is a general way to prove that some statement $S(n)$ about the integer n is true for all $n \geq n_0$ (a certain constant.) It involves two steps:

1. Prove that the statement $S(n)$ is true when n has its smallest value n_0. This is called the basis.
2. We prove that if $S(n_0), \ldots, S(k)$ are true for some $k \geq n_0$, then $S(k+1)$ is also true. This is called induction.

We can illustrate recursive programming design through two examples:
Firstly, imagine you are given an array $A[1, n]$ and you have to reverse the elements in `A`.
The base cases in this scenario are that if the array is empty, we do not have to do anything to reverse it and if there is only one element, we do not have to do anything at all.
We can measure the problem size through n, the number of elements in the array. Notice that if we know how to reverse an array with $n-2$ elements, we know how to reverse an array with n elements. Imagine that the $n-2$ elements are already reversed and sitting in the middle of the array — our only task is to swap the leftmost entry with the rightmost entry. We can do this recursively and solve the problem.
This can be represented as:

```
reverse(A,p,q){
  if(p>=q) return; /* base cases*/
  swap(A[p],A[q]); /*swap the elements A[p] and A[q]*/
  reverse(A,p+1,q-1);
}
```

To reverse the whole array, we need to call $reverse(A, 1, n)$;
You may notice that swap is one of the most important operations in the program — and one that will be executed every time the function is recursively called. Thus, to analyse how many times this instruction is executed, we can set the following recursive equation:

$$[T(n) = \begin{cases} 0, & \text{if } n \leq 1, \\ T(n-2)+1, & \text{if } n > 1. \end{cases}]$$

From telescoping, we ultimately get that the instruction will be executed $T(0 \vee 1) + \lfloor \frac{n}{2} \rfloor$ times. Notice that in this pseudo-code example, we have an instruction named *swap*; if you are writing a C++ program, there is no such function. If you wanted to make the code above executable, you would have to add more detail, but for the effects of algorithm discussion, we tend to use shorthand descriptions similar to this instruction. This makes algorithm presentation more straightforward and not language–specific.

Another example we want to discuss is the following: you are given n characters stored in an array $A[1, \ldots, n]$. Your goal is to generate all the permutations of the n characters.
The base case of this problem is when the array A has only one element: we need to do nothing in that instance!
The recursion strategy is as follows: we divide all permutations in n groups. All permutations in the $i-th$ group are ended with the $i-th$ character. Each group can be formed by permuting the other $(n-1)$ characters and then adding the special one at the end.
Let us state this strategy in code:

```
perm(A,k,n){
/*
Characters stored in A[1,...,n].
Output all permutations of A[1..k] with A[k+1...n] appended.
Chars in array A should be in the same order as it was before perm is
    called.
Assume 1,<=k,<=n. */

  if(k==1)
    output A[1...n];
  else
    for(i=1 to k) do{
    swap(A[i],A[k]);
    perm(A,k-1,n);
    swap(A[i],A[k]); /* restores the array to its original position */
    }
}
```

If we want to solve the original problem, we have to call $perm(A, n, n)$;. After writing this recursive alrogithm, we need to do two tasks: prove its correctness and estimate its running time. When it comes to recursive algorithms, the former is generally analysed by using Mathematical Induction; the latter is done by setting a recurrence equation.
The most essential step in setting a recurrence equation is to identify the fundamental operation(s) within the algorithm — the operations that we perform the most times and that are essential. In the case of the example above, that would be *swap*, which will run k times. This gives us:

$$[f(k) = \begin{cases} 0, & \text{if } n = 1, \\ K[f(k-1)+2] & \text{if } k > 1. \end{cases}]$$

Where f is the amount of times swap will be executed.
Telescoping this expression will eventually lead us to the fact that for $k > 1$, $2(k!) \leq f(k) \leq e(k!)$. Notice that both 2 and $2e$ are constants; both the upper and lower bounds are proportional to $k!$. Because of this fact, we say that $f(k) \in \Theta(k)$.
When it comes to analysing recurrence equations or recurrence functions, it is important to notice that there coould be more than one variable/parameter in

it. Consider *Ackermann's Function*:

$$\begin{aligned} A(0,n) &= n+1 && \text{for } n \geq 0, \\ A(m,0) &= A(m-1,1) && \text{for } m > 0; \\ A(m,n) &= A(m-1, A(m,n-1)) && \text{for } m,n > 0. \end{aligned}$$

2 Algorithms

2.1 Basics

A program is the exact computations you expect a computer to perform. It must be in the exact syntax of some programming language and specify a **finite** sequence of computer instructions.
In contrast, an algorithm also specifies essentially the same computational steps as a specific program written in a specific language (such as Basic, C, Java, Python, …) but its purpose is to communicate a computation procedure to people instead of computers — so, very often, we skip fine details when writing an algorithm. Algorithms often are written in *pseudocode*, a plain–language description of what the program is expected to do, with similarities to the syntax of several programming languages.
Algorithms are programming language independent and machine independent that fulfills two requirements: its operations are well–defined (they can be carried out **unambiguously**) and they halt in a finite amount of time (we can give an upper bound to the algorithm's execution time.)
In an program, we are generally required to handle data. This data is organised using **data sturctures** for efficient retrieval. Thus, a program is said to contain two parts:

- Algorithm: the computational procedure which transforms an input into a desired output as specified by the problem statement.
- Data Structures: the representation, organisation, and storage of information.

Usually, we can come up with several algorithms to tackle a problem. There are two basic metrics that help us identify the most ideal algorithm:

- Problem size: the measure of size, a reflection of the complexity of the problem.
  - Finding the sum of a set of n integers is a problem of size n.
  - Sorting a list of n numbers has a size equal to the amount of entries in the list, n.
  - Multiplying two matrices has a size equal to the dimension of the matrices.
  - Multiplying several matrices has a problem size equal to the dimension of the matrices and the number of the matrices

- Resources: its measure should have, at least, the following properties:
  - Measure the resources we **care about** (e.g. memory space, running time, etc)
  - It should be quantitative
  - It should be a good predictor, so that it can be generalised to different inputs

The primary metric tends to be time. How can we compare the speeds of two algorithms?

The Empirical Approach

Write programs using the different algorithms and time their speeds. This is straightforward but it makes it hard to get a predictor function and has too much noise (such as the input, programmer, language used, etc). When it comes to noise, the input is especially problematic; depending on the it, algorithms may be quicker or slower. Take for instance a search algorithm—it will be significantly quicker if the desired number is in a position favourable to where the algorithm will start searching.

Empirical measuring is used frequently in research, because algorithm analysis can sometimes be extremely difficult and time consuming. When following this approach, it is important to make sure to have a wide sample of unbiased inputs to deal with this possible input–based noise. **The Analytical Approach**

It involves discovering a function of the problem's size that reflects the work the algorithm must do to solve the problem of that size. This is seen as counting some set of operations that the algorithm performs and deriving a rough estimate of the number of such operations required. A simple example that illustrates the idea is the following:

Compute the square of a positive integer n.

Algorithm one:

```
sum = n * n;
```

Algorithm two:

```
sum =0;
for i = 1 to n
    sum = sum+n;
```

Algorithm three:

```
sum = 0;
for i = 1 to n
    for j = 1 to n
        sum++;
```

In order to compare the algorithms, we need to determine the fundamental operations. Here, these will be arithmetic operations (that we assume take constant amount of time). In so doing, we can notice that algorithm one takes one operation, algorithm two takes n operations, and algorithm three takes n^2 operations.

This doesn't seem fair, right? It is likely that a computer performs an increment

faster than a multiplication. Why are we assuming every operation takes the equal amount of constant time? Well, let t_1, t_2, t_3 be the time taken by a computer to perform an increment, an addition, and a multiplication, respectively. The time taken by each algorithm will be then t_3 (algorithm one,) nt_2 (algorithm two,) and n^2t_1 (algorithm three.) As an example, assume that $t_3 = 20t_2 = 200t_1$ (an exaggeration for the purpose of this example.) Then, the time taken by each algorithm will now be $200t_1$ (algorithm one,) $10nt_1$ (algorithm two,) and n^2t_1 (algorithm three.) From this, we can notice that if $1 \leq n \leq 10$, the best algorithm is algorithm three, if $10 < n < 20$ then the best algorithm is algorithm two, and in every other case, the best algorithm is algorithm one.
Thus, **as long as n is large**, the best performing algorithm is the one involving multiplication. Since algorithm analysis is done when the size is large (asymptotically) then we can equate the run time of different operations.

All this covered, how do we solve a computational problem?

1. Recognise the problem. It must be solvable by digital computers.
2. Build an abstract model — the set of operations allowed.
3. We design an algorithm to solve the problem with the model (using only the operations allowed by the mode.)
4. Analyse the algorithm to see what its cost is — the number of operations executed. This gives us the upper bound on the work needed to solve the problem.
5. If possible, analyse the problem itself to see what the minimum amount of work needed to solve it is. This will give us a lower bound to the problem.
6. If we are not satisfied with the algorithm (substantial gap between upper and lower bound) we need to try find a better alternative.
7. Lastly, we build data structures that allow the algorithm to be implemented efficiently. Sometimes, steps three and seven are integrated (algorithm and data structures are designed concurrently).

In cases in which the lower and upper bounds are the same, the algorithm in question is optimal.
Let us see the way we implement those steps in a common problem:
The Towers of Hanoi
There are three diamond needles. On one of these, there are 64 disks. Disks are transfered, one at a time, from one needle to another, in such a way that no disk is ever placed on top of a smaller one. How long will it take to move all of the 64 disks, assuming moving one disk takes $1s$? Let us employ the aforementioned steps:

1. Given three pegs and n disks of different sizes placed in order of size on one peg, transfer the disks from the original peg to another peg with the

constraints that each disk is on a peg, no disk is ever on a smaller disk, and only one disk at a time is moved.

2. Model: the set of operations = {"move one disk from one peg to the other"}
   We choose the measure of the running time of an algorithm for the problem by the number of moves required. The problem size will be n.

3. Let us design the algorithm: our base case will be if there is only one disk ($n = 1$). When this happens, you just move the disk from the source to the destination. The recursive strategy is to carry out three steps: firstly, move the $n - 1$ smaller disks to a buffer peg (assuming we know how to.) Secondly, we move the largest disk to the destination, and lastly, we move $n - 1$ smaller disks to the destination. This can be represented in pseudocode as follows:

```
Hanoi(A,B,C,n){
/* to move n disks from peg A to peg C using peg B as an
    intermediate peg */
    if(n==1)
        move A's top disk to C;
    else{
    Hanoi(A,C,B,n-1);
    move A's top disk to C;
    Hanoi(B,A,C,n-1);
    }
}
```

4. Analysis: is the algorithm correct? We can apply mathematical induction to find this. Next, how many moves does the algorithm take (upper bound?)
   The amount of moves that are executed by the algorithm can be represented in the following recurrence equation:

$$[f(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2f(n-1) + 1 & \text{if } n > 1. \end{cases}]$$

   After telescoping this equation, we find out that for any given n, the number of moves will be $f(n) = 2^n - 1$. Thus, the upper bound of this problem is $2^n - 1$.

5. Now, what is the least number of moves required to solve the problem?
   The most accurate lower bound we can get is $2^n - 1$. This can be proven as follows: let $g(n)$ be the number of moves necessary to solve the problem with n disks (**necessary, not sufficient!**) Before the largest disk is moved the first time, we have taken $g(n - 1)$ moves. This is because essentially, there must be no other disks on top of the largest disk, and one of the pegs must be empty. Before the largest disk is moved for the last time, we need to take an extra move, and then to arrive at the final configuration,

we need to make another $g(n-1)$ moves. Thus, we can conclude that this problem is not solvable with less than $2g(n-1)+1$ for any $n > 1$. Notice that this argument is algorithm–independent.
Setting up a recurrence equation eventually results in that $g(n) \geq 2^n - 1$.

6. Notice that the upper and lower bound are the same. From this, we can conclude that our solution is already optimal — it is not possible to improve it.

If you are curious, the time required to solve this problem is $(2^{64} - 1)s = 18446744073709551615s$ which is roughly 385 billion years!
Even though our problem size was small, it takes a ridiculous amount of time. This is because the solution takes exponential time, which is not ideal. Problems whose lower bound is exponential are called intractable, because they lack a practical solution. If you are faced with this, you have to design an approximation algorithm, a more efficient, non–exponential solution that does not take exponential time.

2.2 Analysing Algorithms

An easy–to–analyse algorithm is one where:

- There is only one problem instance for every problem size
- It is clear which operation(s) we care about in the model

Not every algorithm is like this. What should we do then? Let's illustrate what to do with some examples.
Searching
Given an array of $n \geq 1$ integers and a target x, determine if x is in the array.
Algorithm: let $A[1 \dots n]$ be the array,

```
array_search(A,x,n){
i = 1;
while (x != A[i])
    if(i == n) return ("not found");
    else i++;
return ("found"):
}
```

Let's say we want to analyse this algorithm by counting the amount of operations it performs. It is reasonable to say that the fundamental operation here is the element–to–element comparison. However, the running time of the algorithm depends on the input and the problem instance. For example, if the target element is the first on the array, we will only have to perform one comparison; on the other hand, if the desired valued is the last one in the array, it will take n comparisons. Thus, the cost of the algorithm is not a definite value. In such a case, it is better to express the cost as a range. If the length of array A is n, then the time cost will be in the range $[1, n]$.
The way we set a range is by distinguishing certain cases. Firstly, we would

like to find the *worst cost*, that is, the maximum number of times the algorithm performs the chosen operations. That will give us a **upper bound** of the problem's *worst cost* as a function of the input size. Similarly, the *lower bound* can be the best–case scenario.

- `A`= algorithm
- I_n = set of all possible inputs to A, each of size n
- $f_A(I)$ = the cost of applying algorithm A to problem instance I.
- `WorstCost(A)`= $\max_{I \in I_n} F_A(I)$
- `AvgCost(A)` = $\sum_{I \in I_n} p(I) x F_A(I)$
- if we know $p(I)$ = prob[I occurs],
  $AvgCost(A) = \sum_{I \in I_n} p(I) x f_A(I)$

Sometimes the best–case scenario is trivial — in this example, it is one. Thus, sometimes it is more useful to find the *average cost*; however, $p(I)$ is usually unknown, and even if it is, it is usually hard to calculate. Given an algorithm, it makes sense to **find its worst cost first** and if the problem is important enough, its average cost. In other cases, average cost is better than worst–case scenario for establishing a lower bound; sometimes the worst case is very rare, so it is not a useful representation of the cost of the algorithm (e.g. with hash tables).
There is another important issue to consider: which operations should we count for performance analysis purposes? Take a look at the following two matrix multiplication algorithms:

```
Alg1{ /* given a, b, c, d, e, f, g, h*/
r = a*e + b*f;
s = a*g + b *h;
t = c*e + d* f;
u = c*g + d * h;
output(r,s,t,u);
}
```

This algorithm takes eight multiplications and four additions.

```
Alg2{
/* given a, b, c, d, e, f, g, h */
p1 = a*(g-h);
p2 = (a+b) * h;
p3 = (c+ d) * e;
p4 = d * (f-e);
p5 = (a+d) * (e+h);
p6 = (b-d) * (f+h);
p7 = (a-c) * (e + g);
u = p5 + p1 - p3 - p7;
r = p5 + p4 - p2 + p6;
s = p1 + p2;
t = p3 + p4;
output (r,s,t,u);
}
```

This algorithm takes seven multiplications and eighteen additions.
If we consider multiplication and addition having different costs, which algorithm is better? It depends on which machine you are using — alogorithm comparison would be highly subjective (undesirable). For this reason, algorithm comparison concerns itself about the *growth rate* of the *total number of operations* as a function of the input size *regardless of their types*.
When comparing two algorithms we only compare their orders of growth or complexity. For instance, an algorithm running at $\Theta(n^2)$ is more efficient than one that grows at $\Theta(n^3)$. Growth rate allows us not to worry about the different types of operations having different costs. As long as we are sure that each operation takes a constant amount of time to finish, we only need to count how many operations are needed.
When analysing complexity, we focus on the dominating term, which means we can drop the coefficient that accompanies the dominating term or the lower degree ones.
Notation frequently used:

- Big Theta Θ is a collection of functions that must satisfy the following: $\Theta(g(n)) = \{f(n) : \exists c_1 > 0, c_2 > 0, n_0 > 0$ such that $\forall n \geq n_0, 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)\}$
  This means that $f(n)$ is both upper and lower bounded by a function $g(n)$ multiplied by constants larger than zero. If $f(n) \in \Theta(g(n))$, we say that $g(n)$ is an asymptotically tight bound of $f(n)$. This is valid asymptotically, only for large n (larger than a given n_0).
  Big Theta represents the **average case**.

  Abusing Notation: formally Θ is a set of functions that satisfy some conditions; however, very often we say that a particular function **is** $\Theta(g(n))$. When we say this, it refers to *some function that is in the set* $\Theta(g(n))$. This is just a shorthand way to refer to it. In Computer Science, $\Theta(1)$ is a constant. This technically also includes non–constant values, but convention uses it so.

- Big Oh O gives us an asymptotic upper bound. All functions $f(n) \in O(g(n))$ must satisfy $(g(n)) = \{f(n) : \exists c > 0, n_0 > 0$ such that $\forall n \geq n_0, 0 \leq f(n) \leq c.g(n)\}$
  It represents the **Worst case**

- Big Omega Ω gives us an asymptotic lower bound. All functions $f(n) \in \Omega(g(n))$ must satisfy $\Omega(g(n)) = \{f(n) : \exists c > 0, n_0 > 0$ such that $\forall n \geq n_0, 0 \leq c.g(n) \leq f(n)\}$
  It represents the **best case**

- Litle o
  We use o to denote a proper upperbound. The conditions for the set

$o(g(n))$ are the same as the ones for Big Oh, but the $\leq$ becomes $<$.

- Little omega
  Similarly to little o, ω represents a proper lower bound — switch the $\leq$ for $<$ in the conditions for the set $\Omega(g(n))$

2.3 Calculating Running Time

The following are a series of tricks commonly used when determining the asymptotical complexity of an algorithm:

- Loops:
  The complexity of a loop is **at most** the complexity of the statement inside the loop (including loop condition test) times the number of iterations.
- Consecutive statements:
  The complexity of two consecutive program segments =
  max {complexity of segment 1, complexity of segment 2}
- If–then–else:
  The complexity of `if (condition) S1; else S2` =
  max {complexity of (S1 + cond), complexity of (S2 + cond)}.
- Function call:
  The complexity of a function should be analysed first before computing the complexity of the program fragment containing the call.

Typical Growth Rates

- $O(1)$: constant time
- $O(\log n)$ logarithmic complexity. Commonly seen in programs that reduce a big problem to a small problem, cutting the problem size by a constant factor at each step.
- $O(n)$ is linear complexity. TYpocal for algorithms that do a contstant amount of work on every element.
- $O(n \log n)$ Arises when we break-up a problem into smaller sub–problems and then combining the solutions (*Divide and Conquer*). Some examples are merge sort and quick sort.
- $O(n^2)$ quadratic.
- $O(n^3)$ cubic.
- $O(2^n)$ exponential. Algorithms with exponential complexity are not likely to be practically useful for large inputs.

We say a problem has a tractable solution if at worst it takes $O(n^c)$ for some constant c. However, in practice, n^3 is already considered pretty slow. Some problems only have an exponential solution (i.e. $O(r^n)$) as the most ideal solution. We call these problems **computationally hard problems**, Typically, these problems do not have a polynomial solution, but they cannot be mathematically proven not to have one.

3 Data Strutures

A programming language usually includes certain data types and a collection of operations you can apply on those values. Some also provide additional types such as `strings` (which are usually implemented as character arrays). For these data types, the system hides (*encapsulates*) the details. To the user, strings are just another data type with a set of operations that achieves a specific goal — we do not know how they are implemented or how the operations the data type supports are coded.
We can extend the concept of encapsulation to what is called an abstract data type (ADT). It specifies the information we want to maintain together with a collection of operations that manage the information. To represent an ADT we use data structures connected in various ways. Data structures are essentially collections of variables, possibly of several data types.
Realising an ADT in a program requires us to perform two essential tasks:

- Build data structures to organise and represent information
- Devise algorithms to implement the operations

Aggregate mechanisms are the glue of data structures — they make them more complex. Aggregation allows us to represent complex ADTs.
There are three main types of aggregation:

- Array
- Record Structure
- Pointer

Due to encapsulation, an ADT could be possibly implemented in several ways. Which one should we pick when designing a program? Our goal is devising a way to support all the operations we want to do efficiently and to achieve efficient use of space. Sometimes these two are in conflict with each other — in such a case, we need to assess which one is more important.
Imagine we need to implement an n–by–n plane, where each element is referred to by a pair of indexes and can only take the value 1 or 0. If P is the plane, we want to support the operations "clear(P)", which sets the value of all elements to 0, "state(p, i, j)" which tells you the contents of the element at the index (i,j) and "plot(p,i,j)" which sets the value of (i,j) to 1. We could represent the plane ADT through

- A 2–dimensional array of integers `int P[i][j]`
  This implementation allows us to implement State(P,i,j) by the statement $\in O(1)$

```
if (P[i][j] == 1) then return (1) else(0);
```

  When it comes to space complexity (based on the storage requirement) this implementation $\in \Theta n^2$. This is because every single element will be in the matrix.

- A singly linked list of nodes, each node stores the coordinates of elements whose value is 1.

```
typedef struct point* link;
struct point{int x; int y; link next;};
struct link P;
```

  With this implementation, State(P,i,j) could take, at worst, $\Theta(n^2)$ (the desired pixel is the last element in the last linked list.)
  When it comes to space complexity, this representation $\in \Theta(m)$, where m is the number of elements whose value is one.

As you can see from this example, sometimes no one implementation is best; we must assess based on the situation we are facing.
Let us introduce a very common ADT:
Sets
A set is a collection of elements. Each element may have a *key* identifying it. Some operations that sets are expected to support are:

- `INIT(S)`: initialises a set S (to an empty set)
- `SEARCH(S,k)`: retrieves the element with key k
- `INSERT(S,x)`: inserts element x into the set
- `DELETE(S,x)`: deletes element x from S
- `DELETE_Key(S,k)`: deletes element with key k from set S

If we can compare key values:

- `MINIMUM(S)`: finds the element with the smallest key value
- `MAXIMUM(S)`: finds the element with the largest key value
- `SORT(S)`: sorts the elements according to key value

If the set is ordered:

- `SUCCESSOR(S,x)`: finds the element in S which is ordered next to x
- `PREDECESSOR(S,x)`: finds the element in S which is ordered in front of x.

In a set, we assume that each element has a *key* field. It is a unique identifier.

```
struct element{
    int key;
    ..... /* Other info of an element */
}
```

Another common ADT is the list:

Lists

A list is an **ordered** set of elements. If we let $\{a_0, a_1, a_2, \ldots, a_{n-1}\}$ be a list, it will be of size n. In it, the position of element i will be $i-1$.
Some operations that lists are expected to support are:

- `INIT(L)`
- `SEARCH(L,k)`
- `INSERT(L, i, x)`: insert element x at position i
- `DELETE(L,x)`
- `FIND_ith(L)`: find the i–th element of list L

Some ways of implementing a list are:

- Array Implementation:

```
struct list{
    int length;
    element list_array[MAXLEN]; //max size of the list
}
list L;
```

With this implementation, `INIT(L)`$\in \Theta(1)$.
When it comes to search, the it is normally coded as follows. Note in the worst case, `SEARCH` $\in \Theta(n)$

```
SEARCH(L,k){
    i=0;
    while (i <= L.length -1){
        if (L.list_array[i].key == k)
            break;
        else
            i++;
    }
    if(i>L.length-1) return ("not found");
    else return(i);
}
```

`FIND_ith(L)` takes constant time! (`return L.list_array[i-1];`); however, `INSERT` and `DELETE` are unnecessarily expensive. This is because inserting at the head of the list requires first pushing the entire array down one spot to make room, and deleting an element involves shifting all the elements up by one. The worst case complexity is thus in $\Theta(n)$.

- Linked List
  A linked list is a series of structures (called nodes) that contain each an element and a link to the next node (the `next` pointer). The last node's pointer stores `NULL`. Here, a pointer is a variable that contains the *memory address* where the other data is stored. The value stored in such memory location is accessed via `p->key`.
  With this implementation, `SEARCH` just requires traversing the list via the `next` pointers. It is thus in $O(n)$.
  When it comes to `DELETE`, it can be done by one pointer change (change the `next` element of the previous pointer.) There is, however, one special case we must consider: what happens when we want to delete the first node, which does not have a previous node?
  The most straightforward way to deal with this issue is via adding a *sentinel node* (dummy) at the head of the list. This would alter the linked list implementation to:

```
struct node * node ptr;
struct node{
int key; /* assume key is of type integer */
.....   /* some other fields */
node_ptr next;
}

typedef node_ptr LIST;
```

  Then, the way we implement `INIT(L)` is:

```
INIT(L){
/* first allocate space for the dummy */
L = new node;
L -> next = NULL;
}
/* an empty list is represented by a lone dummy node */
```

  Another function, `IS_EMPTY(L)`, that checks if the list L is empty would be implemented as follows:

```
IS_EMPTY(L){
    if(L->next == NULL)
        return(TRUE);
    else
        return(FALSE);
}
```

  Furthermore, in order to find an element with key k in the list, we implement the following function:

```
SEARCH(L,k){
    node_ptr p;
    p = L -> next; /*p points to the 1st element; NULL if none*/
    while ((p!=NULL)&&(p->key !=k))
        p=p->next;
    return(p);
}
```

Notice that utilising this implementation, `SEARCH` will return `NULL` if there are no matches. Furthermore, the complexity of `SEARCH` is $\in O(size_list)$. `INSERT` requires obtaining a new node and making two pointer changes. It is can be implemented with the code in the segment below, that will insert k after the node pointed to by p:

```
INSERT(p,k){
    node_ptr p1;
    p1 = new node; /* allocate a node */
    p1 -> key = k;
    p1 -> next = p->next;
    p-> next = p1;
}
```

Notice that if we know pointer p, we can achieve insertion in constant time. This is also true if we just insert the element at the head of the list. However, if we want to insert the element at the i–th position, we have to trace down the list through the next $i-1$ pointers to locate the $(i-1)st$ element and insert the new element after it. That takes $O(i)$ time.
At last, when it comes to `DELETE`, we can follow the implementation below for a function that deletes the element *following* the node pointed to by p:

```
DELETE_SUC(p){
    node_ptr p1;
    if (p->next==NULL) return;
    p1 = p->next;
    p->next = p1 -> next;
    delete p1; /* returns the space of the node pointed to by p1
    to the system*/
}
```

If we want to delete the element pointed to by p instead of its successor, we just have to add code that allows us to go through the list and compare pointers:

```
DELETE(L,p){
    node_ptr p1;
    p1 = L;
    while ((p1 -> next != NULL)&&(p1 -> next!= p))
        p1 = p1 ->next;
    DELETE_SUC(p1);
}
```

Notice that this implementation of `DELETE` takes $O(n)$time. We can avoid the search part of this algorithm by adding an extra pointer to the nodes that points to the previous node. This will render our implementation to:

```
DELETE(p){
    p->prev->next = p->next;
    p->next->prev = p-> prev;
    delete p;
}
```

With this implementation of `DELETE`, we should set a dummy tail node (in a similar fashion to the dummy node at the beginning of the list.) A circular doubly–linked list is another useful addition to the structure: it links the tail of the list to the head, and viceversa. If the list is expected to be short, the use of dummy nodes and double links could waste space. Since `DELETE` without these elements will take $O(n)$ time, when we handle a short list it will still be relatively efficient.

Comparing the array and linked list implementations, we know that linked lists are not well–suited for operations that characterise efficient access through indexes in arrays. Linked lists are more appropriate for dynamic data, since they require us rearrange the items often; however, insert and delete require dunamic allocation and deallocation of memory — expensive operations!
Although `INSERT` takes constant time if we do not care which position in the list to insert a new element, `SEARCH` may require traversing the entire list. If this operation will be performed often, we can improve efficiency by ordering the elements in the list according to the key values. This is called a **sorted list**. Once the list is sorted, we can search it with the following implementation:

```
SEARCH_SL(L,k){
    /* assume L is sorted in ascending key value */
    /* assume L is a singly linked list with a dummy header node */
    node_ptr p;
    p = L -> next; /* p points to 1st elements */
                   /*NULL if none */
    while ((p!=NULL)&&(p->key<k))
        p= p -> next;
    if ((p != NULL) && (p->key == k))
        return(p);
    else
        return(NULL);
}
```

Now, let us introduce the **stack ADT**:
A stack is a list with the restriction that inserts and delete can be performed at **one end of the list only** (called the top).
Stacks maintain a last–in–first–out order. Since a stack is just a kind of list, we can implement it using either a singly linked list or using an array. The array implementation is as follows:

```
struct Stack{
    int tos; /* top of the stack */
    element stack_array[max_cap];
}
```

This ADT is expected to support four main operations:

- `INIT`

```
INIT(S) {S.tos = -1;}
```

- `push(x)`: insert at the top of the stack

```
push(S,x){
    if(S.tos == max_cap -1)
        return("Stack overflow!");
    S.tos++;
    S.stack_array[S.tos] = x;
}
```

- `pop()`: return and delete the top element

```
pop(S){
    if(S.tos == -1)
        return("Stack underflow");
    x = S.stack_array[S.tos];
    S.tos--; return(x);
}
```

- `top()`: return the top element

```
top(S){
    return S.stack_array[S.tos];
}
```

Another ADT that we obtain from lists is the **queue**:
A queue is a list where insert is done at one end (tail) and delete is performed at the other end (head,) thus it maintains a first–in–first–out order. It can be implemented the following ways:

- Doubly linked list: we must have a pointer to the head and one to the tail.
- Array:
  The elements in a queue are stored in positions `head, head+1, ..., tail` where we wrap around at the end of the array. Thus, we need two indices *head* and *tail*, plus one count *length* on the queue length.

```
struct Queue{
    int head;
    int tail;
    int length;
    element queue_array[max_cap];
}

INIT(q){
    q.head =1; q.tail = 0;
    q.length = 0;
}
```

Queues are expected to support the following operations (below is their implementation for an array–based queue):

- `enqueue(Q,x)`: add element x to the tail of the queue.

```
enqueue(q,x){
    if(q.length == max_cap)
        return("Queue overflow");
```

```
    tail = (tail + 1) mod max_cap;
    q.queue_array[tail] = x;
    q.length ++;
}
```

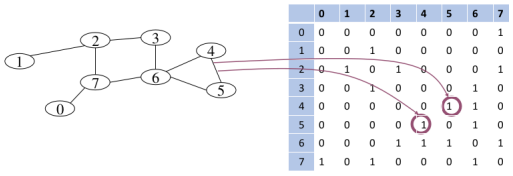
- `dequeue(Q)`: removes and returns the element from the head of the queue.

```
dequeue(q){
    if(q.length == 0)
        return("Queue underflow");
    x = q.queue_array[head];
    head = (head + 1) mod maxcap;
    q.length --;
    return(x);
}
```

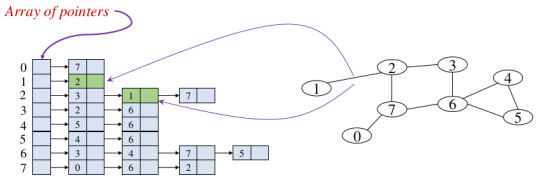
Graph ADT

A graph is a set of vertices and a set of connections (edges) among the vertices. If vertices represent entities or objects, graphs can be used to represent the relationships between each object. Graphs can be bidirectional/undirectonal or directional.

We can represent an undirected graph using a two–dimensional array, called an *adjacency matrix*. In the matrix, the entry $[i, j]$ is set to 1 is vertices i and j are connected, zero otherwise. In undirectional graphs, there will be redundancy, because both entries $[i, j]$ and $[j, i]$ will contain the same information.



Another way to represent a graph is through an adjacency list — an array of linked lists. For each vertex v, we keep a linked list of nodes that are connected to it. Adjacency lists also present redundancy: the link between *one* and *two* will be contained in the linked lists of vertices *one* **and** *two*.



If the graphs are directed, we can simplify the aforementioned structures by reducing the redundancies.

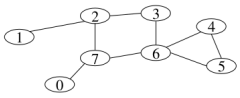
Space Efficiency

The storage requirement of an adjacency matrix is $O(V^2)$ whereas the one for the adjacency list is $O(V + E)$. Thus, if the graph has a big number of edges

(**a dense graph**) the more efficient implementation is the adjacency matrix, because each one of its entries is just a binary bit. Adjacency Matrices are especially efficient for operations that require determining if two vertices are connected. Conversely, if a graph is sparse, adjacency lisks prove more efficient. Lists are more efficient for operations that require processing all the edges in a graph or finding all the neighbours of a vertex.

Breadth–first Search

The way we perform a search in breadth–first order is by starting with a vertex k, visit the vertex that are closest to k, then visit the ones that are another layer away from k, then another, and so on. A vertex that is disconnected from k should not be visited.
Look at the graph below:



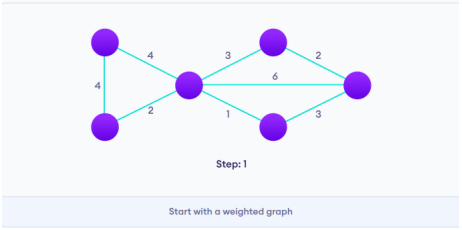
A valid breadth–first order would be $1, 2, 3, 7, 6, 0, 5, 4$. A sample implementation of this searching algorithm is as follows:

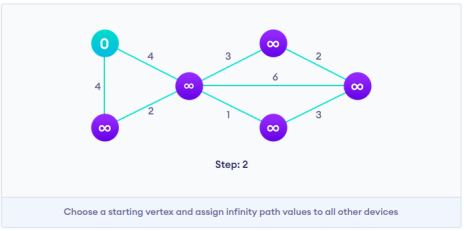
```
BFS(k,n){ /* k = starting vertex; n = number of vertices in the graph
    */
    Queue Q; /* declare a queue Q */
    bit visited[n]; /* declare a bit vector visited */
    INIT(Q); visited[0...n-1] = all 0's;
    enqueue(Q,k);
    while(!empty(Q)){
        i = dequeue(Q);
        if (!visited[i]){
            visited[i] = 1; output(i);
            for each neighbour j of i
                if(!visited[j]) enqueue(Q,j);
        }
    }
}
```

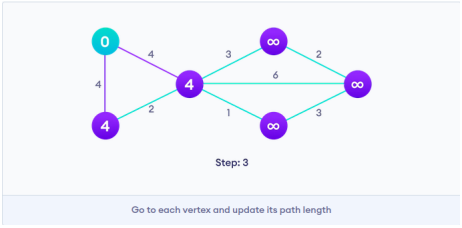
The complexity of this algorithm is $O(V+E)$, where V is the number of vertices and E is the number of edges in the graph. So the complexity is upper–bounded by something proportional to $V + E$.

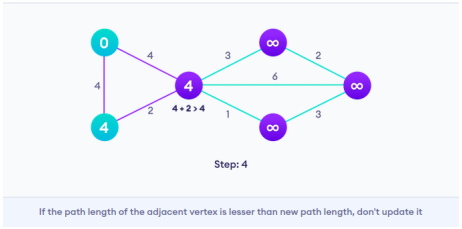
We can also represent data via a weighted graph; each edge is associated with a weight. An interesting thing to ask is the shortest–path problem. The classic algorithm to tackle this is the Dijktra's Algorithm. We use a priority queue instead of a regular queue — we give the highest priority to the element with the largest or smallest weight.
Dijkstra's Algorithm works on the basis that any subpath $B- > D$ of the shortest path $A- > D$ between vertices A and D is also the shortest path between vertices B and D. The algorithm uses this property in the opposite direction, namely by overestimating the distance of each vertex from the starting

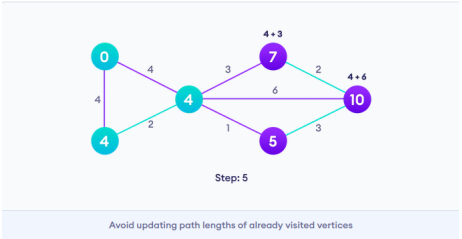
vertex, and then visiting each node and its neighbours to find the shortest subpath to those neighbours. The logic is visualised as follows:

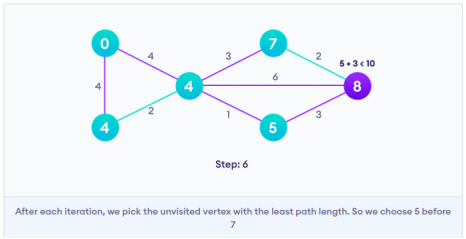


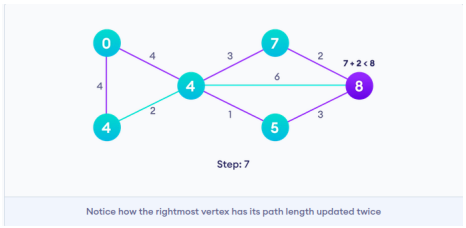


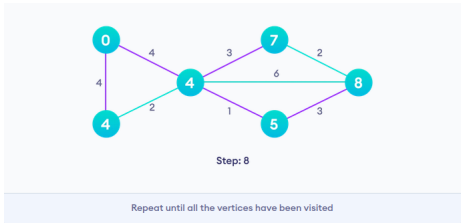












Djikstra's Algorithm can be implemented by storing the path distance of every vertex in an array of size V, where V is the number of vertices. We can also use a minimum priority queue to receive the vertex with the least path distance. In pseudocode:

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Or in a Python implementation:

```
# Dijkstra's Algorithm in Python


import sys

```

```
# Providing the graph
vertices = [[0, 0, 1, 1, 0, 0, 0],
            [0, 0, 1, 0, 0, 1, 0],
            [1, 1, 0, 1, 1, 0, 0],
            [1, 0, 1, 0, 0, 0, 1],
            [0, 0, 1, 0, 0, 1, 0],
            [0, 1, 0, 0, 1, 0, 1],
            [0, 0, 0, 1, 0, 1, 0]]

edges = [[0, 0, 1, 2, 0, 0, 0],
         [0, 0, 2, 0, 0, 3, 0],
         [1, 2, 0, 1, 3, 0, 0],
         [2, 0, 1, 0, 0, 0, 1],
         [0, 0, 3, 0, 0, 2, 0],
         [0, 3, 0, 0, 2, 0, 1],
         [0, 0, 0, 1, 0, 1, 0]]

# Find which vertex is to be visited next
def to_be_visited():
    global visited_and_distance
    v = -10
    for index in range(num_of_vertices):
        if visited_and_distance[index][0] == 0 \
            and (v < 0 or visited_and_distance[index][1] <=
                 visited_and_distance[v][1]):
            v = index
    return v


num_of_vertices = len(vertices[0])

visited_and_distance = [[0, 0]]
for i in range(num_of_vertices-1):
    visited_and_distance.append([0, sys.maxsize])

for vertex in range(num_of_vertices):

    # Find next vertex to be visited
    to_visit = to_be_visited()
    for neighbor_index in range(num_of_vertices):

        # Updating new distances
        if vertices[to_visit][neighbor_index] == 1 and \
                visited_and_distance[neighbor_index][0] == 0:
            new_distance = visited_and_distance[to_visit][1] \
                + edges[to_visit][neighbor_index]
            if visited_and_distance[neighbor_index][1] > new_distance:
                visited_and_distance[neighbor_index][1] = new_distance

        visited_and_distance[to_visit][0] = 1

i = 0

# Printing the distance
for distance in visited_and_distance:
    print("Distance of ", chr(ord('a') + i),
          " from source vertex: ", distance[1])
```

```
63      i = i + 1
```

4 Hashing

A *dictionary* is a set data type that supports three operations: `INSERT`, `DELETE`, and `SEARCH`. Out of these three, the most essential operation is `SEARCH`.
A linked list or an array could be used to implement a dictionary, but searching is not efficient ($\in O(n)$, where n is the number of elements in the set).
To deal with this issue, we introduce **hashing**. It consists of two ideas:

- A table (array) of size m (`T[0..m-1];`).
- A hash function h which maps the key of an element into an integer index in the range of $[0, m-1]$. Thus, an element with key k maps to slot $h(k)$ of the array.

However, this convention faces a problem: collision. What happens if two or more keys have the same hash value? Well, collision handling is defined by the programmer. In general, to handle collision we need a **good hash function** (such that collision does not happen often) and a **collision resolution strategy**.
When it comes to analysing hash tables, there are some important concepts we need to keep in mind: the *load factor* a for a table T is defined as $\frac{n}{m}$, where n is the amount of elements in the table and m is the amount of slots the table has. Intuitively, a represents the average number of nodes stored in a chain.
Let us go through some common ways to approach collision:

- Chaining (Open Hashing):
  It consists in keeping a linked list of all elements that share the same hash value. Using this method, **the load facotr can be larger than one**.
  It is implemented the following way:

```
typedef struct node *node_ptr;
struct node{
    element ele;
    node_ptr next;
}

typedef node_ptr LIST;

LIST T[m]
```

  Furthermore, it is initialised as follows:

```
Chained_Hast_Init(T){
    int i;
    for (i=0; i<m; i++)
        T[i]= NULL;
}
```

  The way we search for a particular key within the list is:

```
Chained_Hast_search(T,k){
    node_ptr p;
    p = T[h(k)];
    while((p!= NULL) && (p->ele.key != k)){
        p = p->next;
    return(p);
    }
}
```

In order to insert an element,

```
Chained_Hash_Insert(T,x){
    node_ptr p;
    p = T[h(x.key)];
    insert x into the linked list pointed to by p;
    /* we might want to perform duplicate check */
}
```

Insertion with this implementation takes constant time ($\in O(1)$). If duplicate check is needed, it is similar to search.
Note that for delete, if we only have the key of the element to be deleted, we have to search for it first. This makes it have a running time similar to searching in a linked list.
So, for searching, the worst case is when all the keys are mapped to the same slot ($\in O(n)$); however, ideally a good hash function should map the keys in the universe as uniformly as possible over the m slots. This ideal hash function is called **simple uniform hashing.** In unsuccessful search using this scheme, the average complexity is $\in O(1+a)$. For a successful search, the average complexity is also $O(1+a)$. We do not drop the 1 in the complexity analysis (even though we usually drop the lower order terms) because we are not sure as to whether a or 1 is the dominating term.

- Open Addressing (Closed Hashing):
  With this scheme, all elements are stored in the hash table itself. Every entry will either contain an element of the dictionary or a special value (`NIL`) that signifies that no element is stored in the slot. Using this method, the load factor **must be less than one.** Ideally, it is below 0.5
  element $T[m]$;
  Compared with chaining, this method avoids pointers. This might be desirable because it saves time (since there is no need for memory allocation and de–allocation) and space. If we are given the same amount of memory, we can afford a larger number of slots under open addressing. This potentially leads to fewer collisions. When collision occurs, we need to try alternative slots until we find an empty slot (in a process called *probing.*) Formally, we consider a probe sequence: $< h(k,0), h(k,1), \ldots, h(k,m-1) >$ where $h(k,i) = [h_1(k) + f(i)] \mod m$ for some function $f()$ of which $f(0) = 0$. Here, $f()$ is the increment function. There is a constraint to consider: the probe sequence should be a permutation of $< 0, 1, \ldots, m-1 >$

so that every slot is included in the sequence.
Insertion can be implemented as follows:

```
Open_Address_Hash_Insert(T, x){
    i = 0; /* collision count */
    k = x.key;
    do {
        j = h(k,i);
        if (T[j].key == NIL || T[j].key == DELETED){
            copy x to T[j]; return;}
        else
            i++;
    }while (i<m)
        return("Hash table overflow!");
}
```

To search for an element, we follow the same probe sequence until the element is found or an empty slot is encountered:

```
Open_Address_Hash_Search(T,k){
    i=0;
    do{
        j = h(k,i);
        if (T[j].key == k)
            return(j);
        i++;
    } while ((T[j].key != NIL) && (i<m))
        return("Not found!");
}
```

To delete an element, we need to follow the same probe sequence until either the element is found or `NIL` is encountered. It is important to notice that we cannot just delete (replace with `NIL`) the object — this will ruin our code for search and delete! We should instead store a special value `DELETED` where the element to be deleted is currently held.

```
Open_Address_Hash_Delete(T,k)
{
    i = 0;
    do{
        j = h(k,i);
        if (T[j].key == k){
                T[j].key = DELETED;
                return;
        }
        i++;
    }
    while ((T[j].key != NIL) && (i<m))
        return("Not Found!");
}
```

When it comes to probing, there are different kinds depending on the increment function $f(i)$ in $h(k,i) = [h_1(k) + f(i)]modm$:

- Linear Probing:
  Here, $f()$ is a linear function of i, typically $f(i) = i$. This amounts to

trying the slots sequentially *with wrap around* in search of an empty slot. The problem with this kind of probing is that **clusters** (blocks of occupied blocks) appear. This makes that after every collision, there will be more and more attempts needed to find another empty spot. The bigger the cluster is, the faster it grows!

- Quadratic Probing:
  Here, $f()$ is a quadratic function of i. For example, $h(k,i) = (h_1(k) + i^2) mod m$. This kind of probing still presents problems however: the probe sequence may not span the whole table. If quadratic probing is used, the table size must be a **prime number** and the table should never be more than half full.
- Double Hashing:
  It utilises two hash functions to define a probe sequence. This is seen as $h(k,i) = (h_1(k) + i(h_2(k))) mod m$; the first hash function is used to determine the initial slot and the second one to determine the increment. This renders clustering unlikely, since it only happens when two keys share the same values for h_1 and h_2. When it comes to setting $h_2(k)$, this function should never evaluate to zero and it should be relatively prime to m — set m to be a prime number and $h_2(k)$ to return values smaller than m.

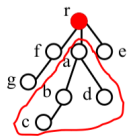
The performance of hashing very much depends on the hash functions. A good h should map the keys in the universe as uniformly as possible over the m slots. h should satisfy the assumption of *simple uniform hashing* and be easy to compute. We are going to cover several common hash functions for keys that are natural numbers:

- Division Method:
  For a key k and a table of size m, $h(k) = k \mod m$. With this method, the choice of m is **very important**. A good hash function should use all the information provided by the keys. Some general rules:
  - m should not be a power of 2. This is because if $m = 2^p$, then $h(k)$ is just the p lowest order bits of k.
  - m should not be a power of 10 if the application deals with decimal numbers.
  - Good values of m are primes.
  - If it is too difficult to find a satisfactory prime, pick m such that it has no prime factors less than 20 (for example, $23x31$)
- Multiplication Method:
  Firstly, we have to pick a constant $0 < A < 1$. Then, we take the fractional part of kxA. Thirdly, we multiply the result by m and take the integer part of the result. This can be represented as $h(k) = int(frac(kxA)xm)$. The choice of A is very important for this method. As a rule of thumb,

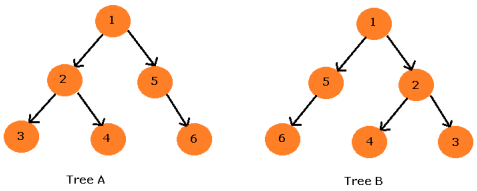
A should be an irrational number. A good choice would be the Golden Ratio.

5 Trees – Introduction

A tree is a connected, undirected graph with no cycles. A tree with no nodes is called a null tree. An undirected graph with no cycles, but *not connected* is called a forest. Trees can be unrooted and rooted; they are rooted when we design one node as the root, i.e. the place the tree originates from.
v_i is a parent of v_j if $(v_i, v_j) \in E$ and v_i appears in the path from root to v_j. Similarly, v_i is an ancestor of v_j if v_i appears in the path from root to v_j. Furthermore, sibligs are nodes with the same parent.
Some other important concepts are leaves (nodes without children) and internal nodes (have both a parent and child.) The root is not an internal node because it has no parent.
We define the degree of a node v in a rooted tree as the number of children that v has. The depth of such node v will then be the length of a path from the root to v. From this concept, we can also obtain a definition for the height of a tree: max (depth of a node in the tree). A subtree of a rooted tree T is a node v of T and all of v's descendants and their edges:

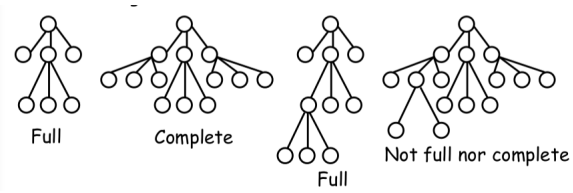


An ordered tree is a rooted tree organised so that the nodes are in some order.



In this example, the left child will be lower than the parent whereas the right child will be larger. Unordered trees are the ones where its nodes do not necessarilly have to follow a rule.
Sometimes, we want to treat left and right children differently. When this is the case, the trees are called positional trees.
A rooted tree is called an m–ary tree if every internal node has no more than m children. M–ary trees are usually treated as positional trees. Now, if $m = 2$, the tree is called a *binary tree*.
An m–ary tree is full if every internal node has exactly m children. It is considered complete if all leaves are of the same depth and all internal nodes are

of degree m. Notice that the number of nodes in a complete m–ary tree with height h will be equal to $\frac{(m^{h+1}-1)}{(m-1)}$.



Trees can be represented as follows:

- Rooted tree with "unbounded" branching:
  The left–child right–sibling representation allows us to save space and also efficiently store the location of every element in the tree.

```
struct node{
    element e;
    node *left-child;
    node *right-sibling;
}
```

- Binary Tree:
  Using pointers, it is implemented like this:

```
struct node{
    element e;
    node *left-child;
    node *right-child;
}
```

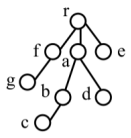
  It is, however, not necessary to use a pointer implementation. We can simulate it by using an array: we use two arrays, one for the left child and another one for the right child. If our parent node is the one with value 3 and it has 1 as a left child and 5 as a right child, then `left_child[3] == 1` and `right_child[3] == 5`. If we are handling a complete binary tree, we can use a single array! The root will be the first element, and if you want to find the children of an element i, you can easily do so. The left child of $T[i] = T[2i+1]$ and the right child will be $T[i] = T[2i+2]$.

Tree specific search:

1. Preorder Traversal: Starting from the root, you visit the node and then repeat the same procedure in each of its subtrees one by one following the order of the subtrees.

```
 Preorder(T){
     if (T == null) return;
     visit(root(T)); //imagine visit() does desired operations on
    the node
     for i = 1 to k do:
         Preorder(T_{i});
```

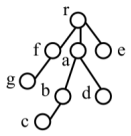
```
        //T_{1}, T_{2}, ..., T_{k} are subtrees of root(T) from left to right
}
```



The preorder of the above tree would be r, f, g, a, b, c, d, e

2. Postorder Traversal: Essentially does the opposite of the preorder; which is to say, it will start from the leaves all the way to the root.

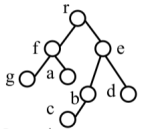
```
Postorder(T){
    if(T == null) return;
    for i = 1 to k do
        Postorder(T_{i});
    visit(root(T));
}
```



The postorder of the above tree would be g, f, c, b, d, a, e, r

3. Inorder Traversal: Visit left child, parent, and then right. The implementation of this search algorithm for a binary tree is the following:

```
Inorder(T){
    if (T == null) return;
    Inorder(T_{1});
    visit(root(T));
    Inorder(T_{r});
    //T_{1} and T_{r} are left and right subtrees of root(T) respectively
}
```



The inorder of the above tree is g, f, a, r, c, b, e, d

The time complexity of all these search algorithms is $\in O(n)$.

5.1 Huffman Code

The regular representation of letters in computers is `ASCII` code: 8 bits per each character. However, there are some letters that appear more frequently than others. Can we not use fewer bits for more frequent letters? For the sake of example, imagine we decide to use the following encoding scheme:

Letter	Frequency	Code
d	2	00
e	1	000
h	2	01
l	2	10
o	5	0
u	1	001
w	1	010
y	1	011

Total: 29 bits

codewords

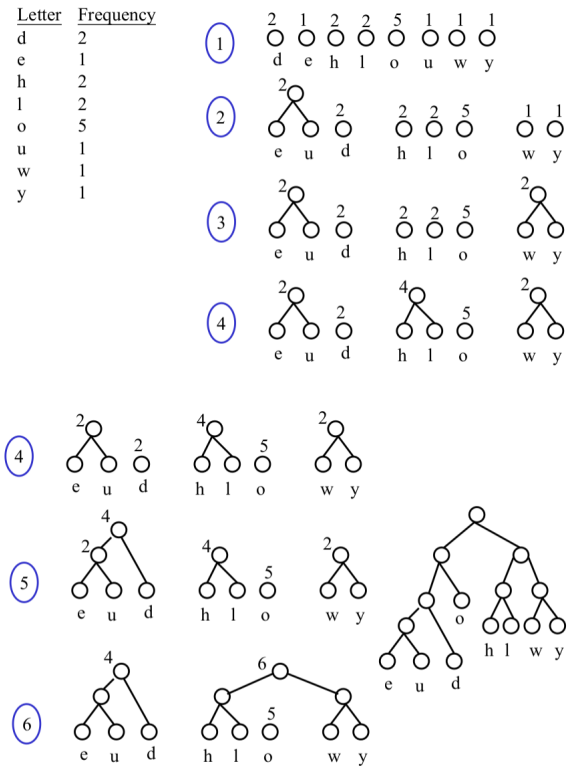
This seems to work, but there are problems when decoding messages because of possible overlap. If we have 00101, it could mean *ohh* or *uh*.
This is solved by using *prefix code*. This consists in making sure that the code for a character will not be the beginning of another code, thus eliminating all ambiguity. A version of the previous table with this scheme applied onto it is:

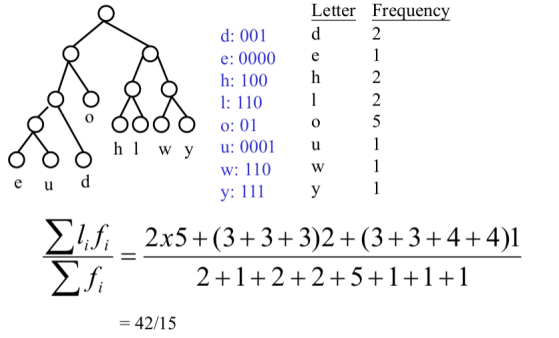
Letter	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

If our input is 1101101101100, we can decode *ebbc* without any ambiguity.
The ideal prefix code, the *Huffman Code*, is the one that minimises $\frac{\sum l_i f_i}{\sum f_1}$, where l_i is the length of the codeword and f_i is its corresponding frequency. How can we create a Huffman Code? We can use a *positional tree* as a representation for it, where less frequent letters are at the lower parts (with greater depths) of the tree. Any left children will be assigned a 0 and right ones will be assigned a 1. In detail, this means that you should **build a binary tree using a bottom–up approach**:

1. Create n trees, each having one node representing one letter/character.
2. Let the weight of a tree be the total frequencies of all symbols represented in its leaves.
3. Repeat the following process until we only have a single tree:
   (a) Pick two trees T_1, T_2 with the smallest weights.
   (b) Create a new tree T with T_1 and T_2 as the left and right subtrees respectively.
   (c) Set the weight of T as weight ofT_1 + weight of T_2.

Take a look at the example below of the application of those steps:





$$\frac{\sum l_i f_i}{\sum f_i} = \frac{2x5 + (3+3+3)2 + (3+3+4+4)1}{2+1+2+2+5+1+1+1}$$

$= 42/15$

6 Binary Search Tree

The searching problem is common in computer science; you are given a set of n elements and an element x that you have to locate within the set (or report as missing.) Let us go through some common search algorithms we have covered:

- Sequential (linear) search:

```
for i = 0 to n-1 {
    if A[i]=x
        return i;
}
return -1; //-1 means not found
```

In this case, both the worst case and the average case $\in O(n)$.

- Hashing:
  The average case complexity for chaining is $\in O(1+\alpha)$, both for successful and unsuccessful searches.
  When it comes to open addressing, the average for an unsuccessful search is $\in O(\frac{1}{1-\alpha})$ whereas the average for a successful one is $\in O(\frac{1}{\alpha}\ln\frac{1}{1-\alpha})$.

- Binary search: Given a *sorted* array, the core idea is to check the middle element and eliminate half of the elements if the target is not found yet.

```
int Search(A,x){
    int lo = 0, hi = n-1, mid;
    while (lo <= hi) do
        mid = floor(lo+hi/2);
        if (x < A[mid])
            hi = mid -1;
        else if (x>A[mid])
            lo = mid+1;
        else return mid;
    return -1;
}
```

Binary search will be $\in O(\log n)$ for both the worst case and the average case.

The best worst case algorithm is binary search; however, insertion and deletion are $\in O(n)$. Not particularly efficient for large n. Can we do better? Well, let us introduce the *binary search tree*.
A binary search tree is a binary tree with keys stored in the nodes which satisfy the *binary–search–tree property*: for every node in the tree, the value of any key in its left subtree is always smaller that its key and the value of any key in its right subtree is always larger than its own. We shall assume that every node has three pointers (parent, left child, right child) and a key stored in it.
We expect binary search trees to support the following operations:

- Minimum:
  The leftmost node will always contain the minimum value in the tree.

```
Tree-Minimum(X){
    while (x.left != null)
        x= x.left;
    return x;
}
```

- Maximum:
  Similarly, the rightmost node will always contain the largest value in the tree.

```
    Tree-Maximum(X){
        while (x.right != null)
            x= x.right;
        return x;
    }
```

  Both minimum and maximum are $\in O(h)$, where h refers to the height of the tree. In the worst case scenario, $h = n$, where n is the number of nodes. However, a complete/balanced binary tree will have $h = O(\log n)$, which will be extremely efficient.

- Search:
  Search is alson $\in O(h)$. It is implemented as follows:

```
        Tree-Search(x,k){
            if (x==null) or (k == key(x))
                return x;
            else
                if (k < key(x))
                    Tree-Search(x.left, k);
                else
                    Tree-Search(x.right, k);
        }
```

- Predecessor:
  Predecessor will return the node whose key is just smaller than that of x or `null` if key(x) is the smallest key.

```
    Tree-Predecessor(x){
        if(x.left != null)
            return Tree-Maximum(x.left);
        y = x.p // parent of x
        while (y != null) and (x = y.left){
            x = y;
            y = y.p;
        }
        return y;
    }
```

- Successor:
  Successor will return the node whose key is just larger than that of x or `null` if key(x) is already the largest key. Summarising this and predecessor, if X has two children its predecessor is the maximum value in its left subtree and its successor the minimum value in its right subtree.

```
    Tree-Successor(x){
        if(x.right != null)
```

```
            return Tree-Minimum(x.right;
        y = x.p // parent of x
        while (y != null) and (x = y.right){
            x = y;
            y = y.p;
        }
        return y;
    }

```

Both predecessor and successor are $\in O(h)$.

- Insert:
  Insertion must maintain the binary–search–tree property. The code for its implementation is

```
    Tree-Insert(T,x){
        y = T; //assume T points to the root
        z = null;
        while (y != null){
            z = y;
            if (x.key < y.key) //insert to left_tree
                y = y.left;
            else //insert to right_tree
                y = y.right;
        }
        x.p = z;
        if (x.key < z.key) // insert as left child
            z.left = x;
        else
            z.right = x;
    }

```

This operation is also $\in O(h)$.

- Delete:
  The core issue with delete is that after performing it, we need to maintain the binary–search–tree property. For easier handling, we will analyse three possible cases for deletion:

  1. The node to be deleted is a leaf:
     This is simple; we just remove the node.

```
        Delete(T, z){ // z points to a lead to be deleted
            if (z.p.left == z) z.p.left = null;
            else z.p.right = null;
            delete z; // delete node and reclaim space
        }

```

  2. The node to be deleted has only one child:
     Here, we will just take the node to be deleted away and we will connect its subtree to the parent.

```
Delete(T,z){
    if (z.left == null) x = z.right;
    else x = z.left;
    if (z.p.left == z) z.p.left = x;
    else z.p.right = x;
    x.p = z.p;
    delete z;
}
```

3. The node to be deleted has two children:
   When this happens, we need to first find the successor of the node we want to delete, switch the successor with the node we want to delete, and then delete our target in the position of the successor. It is worth noting that the successor will never have two children, since by definition it is the leftmost element in the right subtree of our target.

```
Delete(T,z){
    x = Tree-Successor(z);
    z.element = x.element;
    if x has no child:
        delete x as in Case 1;
    else
        delete x as in Case 2;
}
```

Deletion is also $\in O(h)$

The height of a tree is clearly important. Ideally, $h = \log n$. This is called an AVL Tree, which we will cover in the next topic.

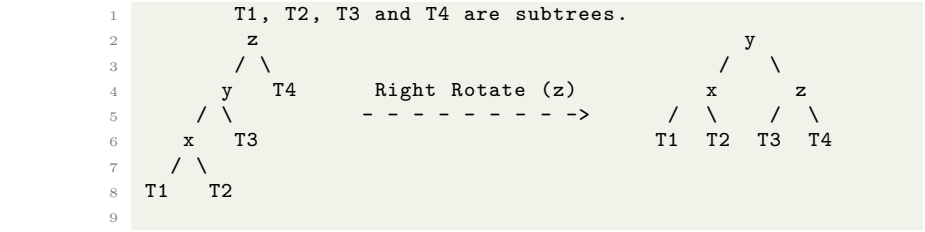
7 Balanced Binary Search Tree – AVL Tree

The core idea behind an AVL Tree is to construct a tree which is balanced (its leaves have more or less the same height) and that maintains this condition after insertion and deletion. Besides the AVL tree, there are other approaches beyond the scope of this class, such as the 2–3 tree, the red–black tree, and the splay tree.
The difference between an AVL tree and these others is that, for every node, the difference between the heights of its left and right subtrees is **at most** 1 (the height of a null tree is -1). The implication of this is that the height of an AVL tree with n nodes is always $O(\log n)$. Implementing this structure presents two big difficulties: making sure the height of a tree with n nodes is $\in O(\log n)$ and performing insertion and deletion of nodes in $O(\log n)$ time. Let us analyse about insertion and deletion for AVL trees:

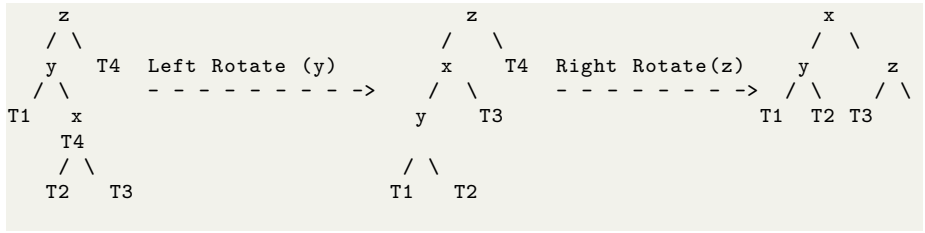
- Insertion:
  We need to maintain the AVL tree property after each insertion. Let the

newly inserted node be w. Firstly, we perform standard BST insert for w. Starting from w, we travel up and find the first unbalanced node. Now, let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z. We need to re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that need to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

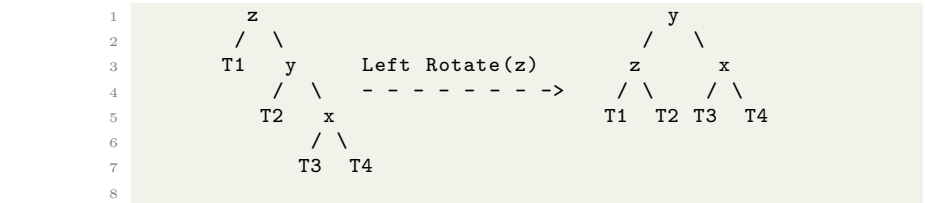
1. y is the left child of z and x is the left child of y (Left Left Case). This case just requires a simple rotation, so it is the easiest one to handle.



2. y is the left child of z and x is the right child of y (Left Right Case). As opposed to the previous case, this case needs two rotations. We perform a left rotation on y, followed by a right rotation on z.

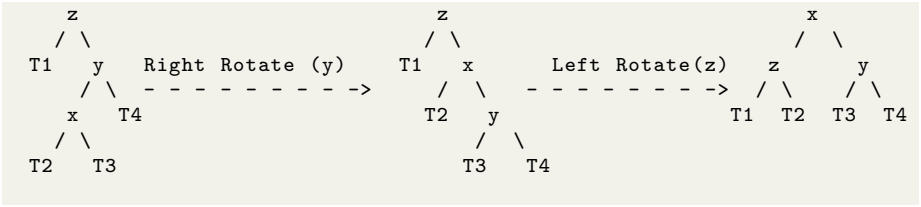


3. y is the right child of z and x is the right child of y (Right Right Case)



This case is relatively straightforward. We just need to perform a left rotation on z.

4. y is the right child of z and x is the left child of y (Right Left Case).



The way we implement the rotation is as follows:
Firstly, we need the following struct:

```
struct node{
    element e;
    int b; //b is the balance factor
    node *left; //-1: right subtree is taller;
    node *right; // 0: equal height
    node *p; // +1: left subtree is taller;
}
```

Then, the insertion procedure is:

1. Insert the node as in the binary search tree
2. Go up to the root along the path from the inserted node, and do the following for each node:
   - Update the value of b
   - Perform rotation to restore balance if the node violates the AVL tree property

- Deletion:
  The procedure for deletion is the following:

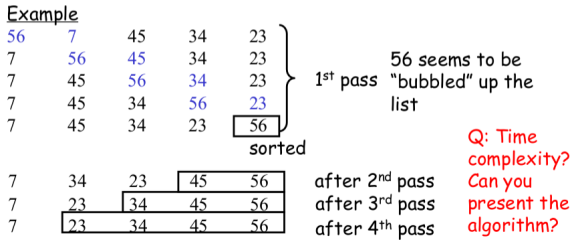
  1. Delete the node as on the binary search tree
  2. Go up to the root along the path from the parent of the node that was just deleted. Do the following for each node:
     - Update the value of b
     - Perform rotation to restore balance if the node violates the AVL tree property.

8 Sorting by Comparisons

A common problem in computer science is, given a sequence of n numbers, to output a permutation of this input so that the numbers are in decreasing (or decreasing) order. We will cover different algorithms that solve this and compare their efficiency.

8.1 Bubble Sort

The core idea of this type of sorting is that, at each pass, the algorithm will scan the numbers from left to right. If the left number is higher than the right number, it will swap them. This is an extremely inneficient method of sorting — its use is mostly restricted to introductory coding courses. For instance:



It can be implemented as follows:

```
Initialize n = Length of Array
BubbleSort(Array, n)
{
    for i = 0 to n-2
    {
        for j = 0 to n-2
        {
            if Array[j] > Array[j+1]
            {
                swap(Array[j], Array[j+1])
            }
        }
    }
}
```

Bubble sort is an *in–place* sorting algorithms. This means that very little or none extra storage is needed to achieve successful sorting. Since Bubble Sort iterates the full array for every element, it has a time complexity of $O(n^2)$.

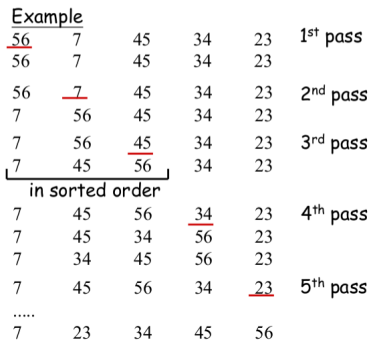
8.2 Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part. This sorting algorithm is preferably used on small n. It is also more effective on data that is already partially sorted. As with Bubble Sort, insertion sort is an in–place sorting algorithm. It is implemented as follows:

```
procedure insertionSort(A: list of sortable items)
   n = length(A)
   for i = 1 to n - 1 do
       j = i
       while j > 0 and A[j-1] > A[j] do
           swap(A[j], A[j-1])
```

```
            j = j - 1
        end while
    end for
end procedure
```

A sample execution of insertion sort is:



The average complexity of this algorithm is $O(n^2)$. Its best case complexity is $O(n)$.

8.3 Selection Sort

Selection Sort works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list. The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted portion. This process is repeated for the remaining unsorted portion of the list until the entire list is sorted.

The algorithm maintains two subarrays in a given array: the subarray which is already sorted and the remaining subarray that is unsorted. In every iteration of Selection Sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the beginning of the sorted subarray. After every iteration, the sorted subarray's size increases by one and the unsorted subarray's size decreases by one. After the Nth (size of the array) iteration, we will get a sorted array.

Selection Sort is implemented the following way:

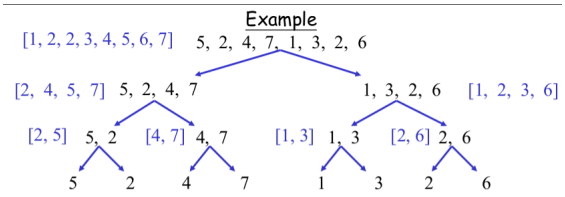
```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
```

```
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for

end procedure
```

The time complexity of this algorithm is $O(n^2)$. As with the two previous algorithms, Selection Sort is in–place.

8.4 Merge Sort

It consists dividing an array of numbers into two lists of roughly equal lengths, sorting them recursively using Merge Sort, and then merging the two sorted lists. Let us visualise this procedure:



The way this can be implemented is as follows:

```
MergeSort(A,p,r){ //Merge sort A[p] to A[r]
    if (p<r){ //terminating condition for recursion
        q = floor((p+r)+2); //divide into roughly equal length lists
        MergeSort(A,p,q);
        MergeSort(A, q+1, r);
        Merge(A, p, q, r) //Merge A[p].. A[q] and A[q+1]...A[r]
    }
}
```

As you might be able to tell by the image of the procedure, the time complexity of this algorithm $\in \Theta(n \log n)$. Good, right? Well, the problem is that this algorithm is **not** in place: it requires a lot of extra space for the merging of the different sorted lists. Therefore, it is not ideal.

8.5 Heap Sort

A **max**–heap is a binary tree (NOT a binary search tree) that satisfies two properties:

1. The value for a node $\geq$ the value of any of its children.
2. If the height of the tree is h, then there are 2^i nodes with depth i ($i < h$) and the nodes at depth h are packed **from the left.**

When it comes to max–heaps, the node that stores the maximum value will clearly be the root. Similarly, if the height of a heap is h, the maximum number of nodes in a heap will be $2^{h+1}-1$. The minimum number of nodes is 2^h. If we want to find the minimum value in this structure, we just need to examine the leaves.
A heap can be represented as an array. If this is the case, then

- The left child of $A[i]$ is $A[2i]$
- The right child of $A[i]$ is $A[2i+1]$
- The parent of $A[i]$ is $A[floor(\frac{i}{2})]$

Since, unlike Binary Search Trees, there is no difference in condition between left and right child, insertion is more straightforward. If the newly–inserted value violates the max–heap, we just have to compare it with its parent (in a loop) and switch it until the parent is larger than it. It can be implemented as follows:

```
Insert(A,x){ //A is the array storing the heap
    A[size+1] = x; //Put x at the end of the array
    size ++; //update heap size
    i = size;
    while (i>1) and (A[i]>A[floor(i/2)]){ //check heap property with
    parent
        swap(A[i], A[floor(i/2)]);
        i=floor(i/2)
    }
}
```

Building a heap:

- Top–down approach:
  Given an array of elements, we divide such an array in two parts: sorted and unsorted. We add the elements one–by–one to the sorted array and then adjust its position by moving it up the heap as much as needed. For example: $1, 3, 2, 5, 4$
  $1, ||, 3, 2, 5, 4$
  $1, 3, ||, 2, 5, 4$
  $3, 1, ||, 2, 5, 4$ *Heapify the added element by comparing it with its parent*
  $3, 1, 2, ||, 5, 4$ *Add the third element to the sorted array.*
  $3, 1, 2, ||, 5, 4$ $2 < 3$*; heap already sorted*
  $3, 1, 2, 5, ||, 4$
  $3, 5, 2, 1, ||, 4$ *Swap 5 with its parent because parent smaller than child*
  $5, 3, 2, 1, ||, 4$ *Swap 5 with its parent because parent smaller than child*
  $5, 3, 2, 1, 4$
  $5, 4, 2, 1, 3$ *Compare 4 with its parent and swap. Done!*
  This can be implemented as follows:

```
    void buildHeapifyUp(vector<int> &arr){
    for(int i=0; i<arr.size(); i++){
        int j = i;
```

```
        // while root is smaller than child
        // swap root with child
        while( j>0 && arr[j/2]<arr[j]){
            swap(arr[j/2],arr[j]);
            j = j/2;
        }
    }
}

```

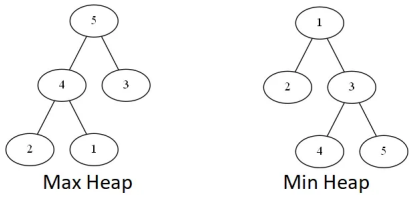
The time complexity of this method is $\in O(n \log n)$.

- Bottom–up approach:
  In the previous method, when we add a new node, we tend to shift it upwards to make a heap. If we try to see the number of swaps, the deepest layer with the most nodes needs the most swaps. If we move the nodes downward instead, the number of swaps decreases — the first layer, the one with the least nodes, is the one that requires the most swaps under this scheme! For example:
  $1, 3, 2, 5, 4$ *We start from the first node with a child that has the largest array index; index = len/2 -1*
  $1, 5, 2, 3, 4$ *Starting from index 1, compare with two children. Swipe with largest (5)*
  $5, 1, 2, 3, 4$ *We move to index 0. Compare it with its children and swipe with largest*
  $5, 4, 2, 3, 1$ *Compare 1 with its two children. Swap with the largest and we finish!*
  This can be implemented as follows:

```
    void buildHeapifyDown(vector<int> &arr){
    for(int i=arr.size()/2-1; i>=0; i--){
        int j = i;
        // while root is smaller than child
        // swap root with child
        // and then heapify down the new child
        while( 2*j+1 < arr.size()){
             int l=2*j+1, r=2*j+2;
             if(r<n && arr[r]>arr[j]){
                 swap(arr[r],arr[j]);
                 j=r;
             }
             else if(arr[l]>arr[i]){
                 swap(arr[l],arr[j]);
                 j=l;
             }
             else break;
        }
    }
}

```

The complexity of this method is just $\in O(n)$!

A **min**–heap satisfies the same conditions, except that the value for a node $\leq$ the value of any of its children.



Now that we know enough of heaps, let us cover Heap Sort — we must take advantage of the fact that the heap allows us to extract numbers easily. Hence, it is implemented as follows:

```
1        Extract-Max(A){
2        /* Copy the value from the root */
3        /* Exchange the values of the right-most leaf and the root */
4        /* Delete the right-most leaf */
5        /* Rebalance the heap; swap the parent with the larger child if
     parent < children */
6        }
7        HeapSort(A){
8            buildHeapifyDown(A);
9            for (i = 1 to n-1){
10               temp = Extract-Max(A);
11               A[size] = temp;
12               size --;
13           }
14       }
15
```

Alternatively, this can be implemented as:

```
1    void heapSort(vector<int> &arr){
2        heapify(arr);
3        int len=arr.size(),l,r;
4        for(int i=0; i<arr.size(); i++){
5            swap(arr[0],arr[len-1]);
6            len--;
7            int j=0;
8            while( 2*j+1 < len){
9                int l=2*j+1, r=2*j+2;
10               if(r<len && arr[r]>arr[j] && arr[r]>arr[l]){
11                   swap(arr[r],arr[j]);
12                   j=r;
13               }
14               else if(arr[l]>arr[j]){
15                   swap(arr[l],arr[j]);
16                   j=l;
17               }
18               else break;
19           }
20       }
21       for(int i=0;i<arr.size();i++) cout<<arr[i]<<" ";
22       cout<<endl;
```

```
23      }
24
```

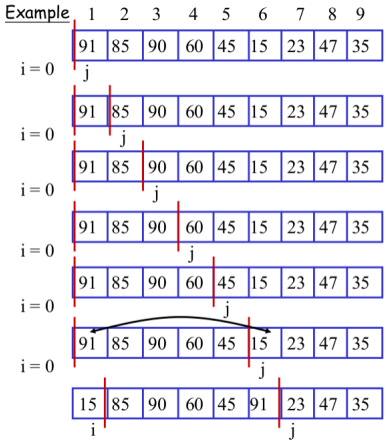
Its worst case is $\in O(n \log n)$. It is an in–place algorithm.

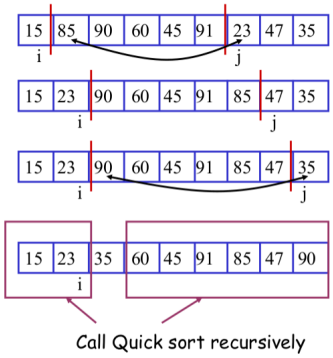
8.6 Quick Sort

The idea behind quick sort is attempting to implement Merge Sort (dividing the numbers into two lists) while avoiding the work implied in merging the lists. In Merge Sort, we divide the input into two lists by cutting (roughly) in the middle, and then we sort. In quick sort, we do something more:

1. Pick an element A, called pivot v
2. Move elements $\leq v$ to the left of v
3. Move elements $> v$ to the right of v

We will use two pointers, i delimiting where the numbers smaller or equal to v end, and j, delimiting where the numbers larger than v end. If the number is larger than v, just move the pointer j. If the number is smaller or equal, swap the element at i with the current element, and move i and j by one. Lastly, if j = v, swap the element being pointed at by i (first one in the larger element list) with v.





This can be implemented the following way:

```
Partition(A,p,r){ //partition A[p..r] using A[r] as pivot
    v = A[r];
    i = p-1;
    for (j = p to r-1){
        if (A[j]<=v){
            swap(A[j],A[i+1]);
            i = i+1;
        }
    }swap(A[r],A[i+1]); //for the pivot
    return i+1; //return position of the pivot
}

QuickSort(A,p,r){ //sort A[p..r]
   if(p<r){ //otherwise done!
       q=Partition(A,p,r);
       QuickSort(A,p,q-1);
       QuickSort(A,q+1,r);
   }
}
```

Its worst case is $\in O(n^2)$; its average case is $\in O(n \log n)$. It is an in–place algorithm.

In short, all of this can be summarised as:

	Worst Case	In-Place	Average Case	Best Case
Bubble Sort	$O(n^2)$	Yes	$O(n^2)$	$O(n)$ (Input Already Sorted, else $O(n^2)$)
Insertion Sort	$O(n^2)$	Yes	$O(n^2)$	$O(n)$ (Input Already Sorted, else $O(n^2)$)
Selection Sort	$O(n^2)$	Yes	$O(n^2)$	$O(n)$ (Input Already Sorted, else $O(n^2)$)
Merge Sort	$O(n \log n)$	No	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	Yes	$O(n \log n)$	$O(n)$
Quick Sort	$O(n^2)$	Yes	$O(n \log n)$	$O(n \log n)$

9 Sorting in Linear Time

All the previous sorting methods have features in common:

- They make no assumptions on the values of the numbers. All the algorithms are based in the comparisons between elements.
- The best algorithm we learned runs in $O(n \log n)$ in the worst case. There are examples that require $\Omega(n \log n)$ time.

It is actually not possible to have a better–performing algorithm that is based on element comparison.
However, if the algorithm is not based in comparison between numbers, it is possible for it to be faster. We will cover three such algorithms.

9.1 Counting Sort

We must assume that the numbers being sorted are **integers** in the range 0 to k. The idea is that, if I know that m numbers are less than x, I should place in a specific position in the sorted sequence (position $m+1$?)
We implement it by using two arrays, $C[0, \ldots, k]$ (working storage) and $B[1, \ldots, n]$ (output), as follows:

1. Scan A (input array) and store the number of occurrences of i in $C[i]$

```
    for (i=1 to n){
        C[A[i]] = C[A[i]] + 1 //assume C has been initialised to
    0;
    }

```

   This is $\in O(n)$.

2. Determine the number of elements $\leq i$ for all $i \geq 1$ based on C

```
    for (i=1 to k){
        C[i] = C[i] + C[i-1];
    }

```

   This is $\in O(k)$.

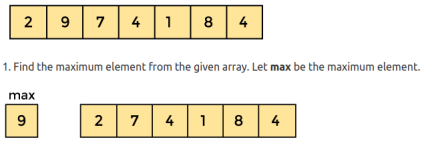
3. Place $A[j]$ in the appropriate position using information in C.

```
    for (i = n downto 1){
        B[C[A[i]]] = A[i];
        C[A[i]] = C[A[i]] - 1;
    }

```

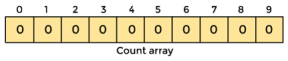
   This is $\in O(n)$

Thus, the total time complexity of this algorithm $\in O(n+k)$; if $k = O(n)$, then it is $\in O(n)$. It is clearly not an in–place algorithm, since it requires of storage, intermediate arrays. Let us look at the following example:

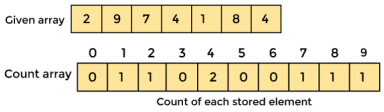
1. Find out the maximum element of the given array.



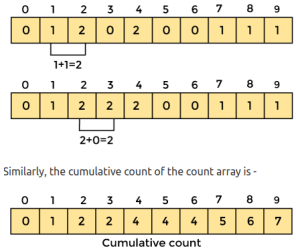
2. Initialise an array of length `max+1` with all elements 0. This array is used for storing the count of the elements in the array.



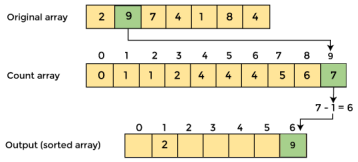
3. Store the count of each element at their respective index in `count` array.



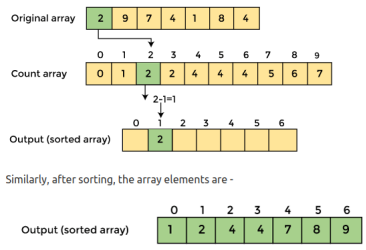
4. Store the cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.



5. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated.



6. After placing each element at its correct position, decrease its count by one



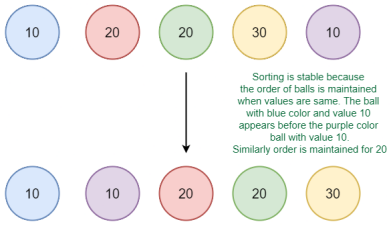
The C++ of this is:

```
#include <iostream>
using namespace std;

int getMax(int a[], int n) {
int max = a[0];
for(int i = 1; i<n; i++) {
    if(a[i] > max)
        max = a[i];
     }
    return max; //maximum element from the array
    }

void countSort(int a[], int n) // function to perform counting sort
{
    int output[n+1];
    int max = getMax(a, n);
    int count[max+1]; //create count array with size [max+1]

    for (int i = 0; i <= max; ++i)
    {
        count[i] = 0; // Initialize count array with all zeros
    }

    for (int i = 0; i < n; i++) // Store the count of each element
    {
        count[a[i]]++;
    }

    for(int i = 1; i<=max; i++)
        count[i] += count[i-1]; //find cumulative frequency

    /* This loop will find the index of each element of the original
    array in count array, and
     place the elements in output array*/
    for (int i = n - 1; i >= 0; i--) {
        output[count[a[i]] - 1] = a[i];
        count[a[i]]--; // decrease count for same numbers
    }

    for(int i = 0; i<n; i++) {
        a[i] = output[i]; //store the sorted elements into main array
       }
    }

void printArr(int a[], int n) /* function to print the array */
```

```
{
    int i;
    for (i = 0; i < n; i++)
        cout<<a[i]<<" ";
}

int main() {
    int a[] = { 31, 11, 42, 7, 30, 11 };
    int n = sizeof(a)/sizeof(a[0]);
    cout<<"Before sorting array elements are - \n";
    printArr(a, n);
    countSort(a, n);
    cout<<"\nAfter sorting array elements are - \n";
    printArr(a, n);
    return 0;
}
```

9.2 Radix Sort

Radix Sort sorts a set of integers by one digit in each pass. If the digits have d digits, d passes are sufficient to sort them. At each pass, we only look at one digit, starting from the least significant digit (the rightmost one.) For each pass, any available sorting algorithm can be used, as long as it maintains the relative order of two elements with the same digit being considered, i.e. a stable sorting algorithm. For instance, the following is a stable algorithm:



Out of the ones covered, *heap sort* and *quick sort* are unstable.
Radix Sort can be implemented through:

```
RadixSort(A,d){ //each integer has d digits
   for i = 1 to d{ //let digit one be the rightmost digit
       use (stable sort) to sort A on digit i;
   }
}
```

Assuming that each digit can have k different values, the overall complexity of Radix Sort is $O(d(complexity_of_sorting_algorithm))$. If we use counting sort, it would be $O(d(n+k))$. Furthermore, if d is a constant and $k = O(n)$, Radix Sort runs in linear time!

9.3 Bucket Sort

Bucket sort is a sorting algorithm that involves creating a number of buckets and distributing the elements of an array into these buckets. Each bucket is

then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. Bucket sort is useful when the input is uniformly distributed over a range. For example, consider the following problem: Given an array of n numbers, each of which is between 0 and 1, design an algorithm to sort them in linear time.