Recursive programming:

- Recursive function: the definition contains a call to itself.
- Useful when we can break the task into smaller similar subtasks.
- Don’t forget! A recursive definition requires a terminating condition to prevent infinite recursion.

Iteration:

- while, for, do LOOPs

Elementary operations:

- addition
- multiplication
- comparison
- memory write, swapping

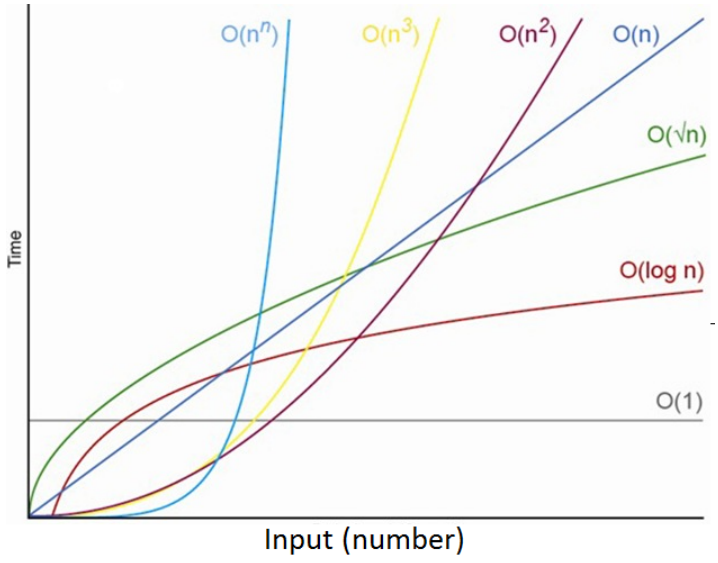
Input “size”:

- length of an array, e.g. for searching
- largeness of an input scalar, e.g. for primality test, or integer factorization
- it can depend on multiple input factors e.g. shortest path in a graph

Time Complexity: (How the elapsed time, T of an algorithm, varies as the input “size” varies.)

- Complexity analysis – finding a mathematical expression for measuring the growth-rate of the the number of elementary operations needed for an algorithm to finish.
  Ignoring low-level details such as the implementation, programming language, hardware, the instruction set of the CPU, optimization, etc.

Example: Complexity of Sequential and Binary Search
How many comparisons are necessary for finding an element in a n-element array?
Big O notation (asymptotic and upper bound): Sequential search: O(n), Binary search: O(log (n))



Data Structures:

- A **stack** is a container of objects that are inserted and removed according to the Last-In First-Out (LIFO) principle.

Stack operations are : createStack, emptyStack, push, pop, top.
Stack Application Examples: "Undo", Parsing, ..

- A **queue** is a container of objects that are inserted and removed according to the First-In First-Out (LFIFO) principle.

- **Structure** and structure **array**.

Application Examples: Store and retrieve data

- **tree**

Sorting Algorithms are algorithms that put elements (permutation) of a list in a certain order.
Properties: (time) complexity, stability, memory usage, in-place, adaptive, online, …

- **Selection** sort (inefficient on large lists)
  * find the smallest number in the unsorted part of the list
  * swap the first element with the smallest in the unsorted part
  * the sorted part of the list expands with one element, the unsorted reduces with one element
  * if the unsorted part isn't empty, use the algorithm again

- **Insertion** sort (efficient for small lists and mostly sorted lists)
  *create an empty sorted list
  *take elements from the list one by one, for each:
  *find the correct position in the sorted list
  *insert it in its correct position into the sorted list

- Divide and conquer (Mergesort and quicksort)
  if length of list is 1 or 0:
  * the list is sorted if length of list is greater than 1
  * Partition the list into a lower and a higher list
  * Sort the lower list
  * Sort the higher list
  * Combine the sorted lower and sorted higher lists

Mergesort (von Neumann, 1945):
*Divide the list into two sublists of equal size and sort them separately.
*Merge the two sorted lists into one.

Quicksort (Hoare, 1960):
*Elements less than the pivot go to the lower list.
*Elements greater than the pivot go to the higher list. Apply Quicksort to the sub-lists.

- **Radixsort** (A non-comparison-based sorting algorithm!)
  *Place the numbers in boxes (0 - 9) with respect to the last digit:
  *Put the sublists together. Start with box 0 and keep the internal order!
- **Bubble** sort (Demuth, 1956)
  * scan the unsorted list from start to end
  * if two adjacent numbers aren't in order, swap

* the largest number in the unsorted list will bubble up to the sorted list
* the sorted list expands with one element, the unsorted reduces with one element
* if the unsorted list isn't empty, use the algorithm again

Efficient implementations generally use a hybrid algorithm, combining an asymptotically efficient algorithms for the overall sort with insertion sort for small lists at the bottom of the recursion.

Up and Running with C – What Do We Need?

- Declaration of variables
- Basic types (`%d int, %f float, %lf double, %c char, ...`)
- Aggregates of basic types (arrays, martices, structures, linked lists, trees)
- Type conversion (`char < int < long < float < double` , BUT sometimes it is advisable to do an explicit casting; `(type) expression`)
- Formatted input `(#stdio.h` for use of `scanf(const char *format, ...);`)
- Formatted output (`#stdio.h` for use of `int fprintf(FILE *stream, const char *format, ...);`)
- Arithmetic expressions (`+, - , * , / , =`)
- Compound assignments (`-= , *= , /=, ++i or i++`)
- Logical expressions (`<, >, <=, >=, ==, !, !=, &&, ||`)

- Selection statements (`if(expression)` statement `else` statement)
- The `switch` statement with `case` and `default` (For simple use of `switch` the last statement in each case group should be `break;`)
- Iteration statements (loops: `while, do, for`)

- Functions in C can be viewed as small programs on their own. (recursive functions,)
  Passing arguments by value or reference? If an argument is passed by reference (`*`pointer, `&`addres), any changes made to the parameter inside the function will affect the value of the argument outside.
  Give one example of a problem where one can use call-by-reference but NOT call-by-value to solve the problem. (bank state account manipulation)

Variable scope: local and file(global) scope.
Variables declared inside a block (compound statement) are visible to the end of the block and have local scope. Local scopes can be nested. Variables declared outside any function definition are visible to the end of the file and have file scope. These global variables should be avoided if possible (simplifies debugging).

Pointer variables:
Pointer variables `type *x;` are intended for pointing at locations in memory rather than storing values. They can point to any object of a specific type, including basic types, arrays, pointers and functions. Initially `x` points to `NULL` (`=0`, no address); `x` can point to any variable of correct type.
How are pointers really used?

- Refer to new memory allocated during program execution
- Refer and share large data structures without making a copy of the structures
- Specify relationships among data – linked lists, trees, etc.

Arrays as function arguments:

- Arrays can be used as function arguments but a function cannot return an array!
- Arrays are always passed by reference, that is, all changes to the parameter will also affect the argument.
  `return-type function-name(int length, type *parameter);`
- We can/must also supply the function information on the length of the array. For one dimensional arrays, a function declaration may look like.

Pointers in structures :
Structures are collections of values (members), possibly of different types, used for storing related data. The value of a member is accessed through `identifier.member`.
A structure can be associated with a tag `struct tag { ...};`using the tag, a structure variable can be declared as: `struct tag name;`
A typedef can be used to simplify the declaration for a struct or pointer type, and to eliminate the need for the `struct` key word `typedef struct my_tag {int i; ...} my_type;`

It is sometimes necessary to have pointers to structures. The members of a structure can be accessed by either of `(*pointer_to_struct).member` or `pointer_to_struct->member.`

Self-referential structures are structures with a pointer member that points to the structure itself. With structures like these, one can create dynamic data types like linked list, stack and queue.

Linked List:
Sequence of elements called nodes (data structures) that have data and a link (pointer) that can reference nodes or null (typically end of list).

- Advantage: Inserts/Deletes in constant time
- Disadvantage: Random access in linear time (vectors are better, remember why?), slower for sequential traversal (cache)

Dynamic memory allocation:
Memory can be allocated during program execution using the functions `malloc` and `calloc` (`stdlib.h`) which allocate memory and return a pointer to the memory block, or `NULL` if not enough memory is available.
When a memory block is no longer needed, it should be deallocated so that it can be reused for other purposes.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

Abstract Data Types (ADTs)

An abstract data type (ADT) is a model for:

- a certain class of data structures that have similar behavior, or certain data types that have similar semantics.
- describing operations and constraints.

Examples: Stack (LIFO: Last-In, First-Out), Queue (FIFO: First-In, First-Out), Tree

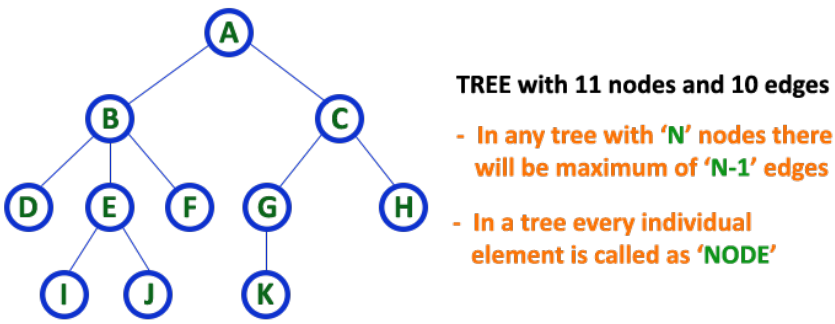
What are the differences between a queue and a tree?

Key Differences Between Linear and Non-linear Data Structure:

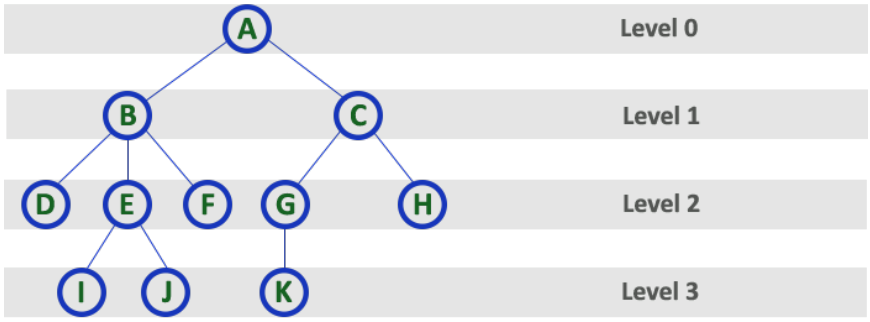
- In the linear data structure, the data is organized in a linear order in which elements are linked one after the other. As against, in the non-linear data structure the data elements are not stored in a sequential manner rather the elements are hierarchically related.
- The traversing of data in the linear data structure is easy as it can make all the data elements to be traversed in one go, but at a time only one element is directly reachable. On the contrary, in the non-linear data structure, the nodes are not visited sequentially and cannot be traversed in one go.
- Data elements are adjacently attached in the linear data structure, which means only two elements can be linked to two other elements while this is not the case in the non-linear data structure where one data element can be connected to numerous other elements.
- The linear data structures are easily implemented relative to the non-linear data structure.
- A single level of elements is incorporated in the linear data structure. Conversely, non-linear data structure involves multiple levels.
- The memory is utilized efficiently in the non-linear data structure where linear data structure tends to waste the memory.

Tree Terminology:

- **Root** Node – the origin of tree data structure. In any tree, there must be only one root node.
- **Edge** – the connecting link between any two nodes. In a tree with 'N' number of nodes there will be 'N-1' number of edges.
- Parent Node – a node which has branch from it to any other node downwards (away from the root).
- Child Node – the node which has a link from its parent node is called a child node. Any parent node can have any number of children (unless it is a binary/ternary/quaternary/... tree!). All nodes except the root are child nodes.
- Siblings – nodes that share the same parent.
- **Leaf** Node – a node which does not have children.



- Level – in a tree structure, each step from top to bottom is called a Level. The Level count usually starts with ’0’ and is incremented by 1 at each level (Step).

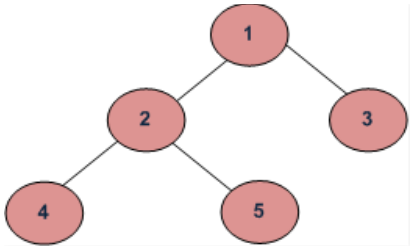


- Path (between two Nodes) – a sequence of nodes and edges connecting a node with a descendant node.
- Length of a Path – total number of edges in that path. In the example below the path A - B - E - J has length 3.

Binary Tree:
Each node has at most two children, which are referred to as the left child and the right child:
Full Binary Tree (full nodes), Complete Binary Tree (full levels), Perfect Binary Tree (full internal nodes & all leaves are at the same level.)

Traversing a binary tree:

- Breadth First or Level Order Traversal:
  Example: 1 2 3 4 5
- Depth First Traversal:
  Preorder (Root, Left, Right) ; 1 2 4 5 3
  Inorder (Left, Root, Right) ; 4 2 5 1 3
  Postorder (Left, Right, Root) ; 4 5 2 3 1



<u>When are trees used?</u>

- Data compression
- Databases
- File system
- Compiler uses syntax trees
- Chess program
- Decision trees
- Binary heap* / Heap sort

*A binary heap is a binary tree with two additional constraints:

- Heap property: All nodes are greater than or equal to each of its children
- Shape property: All levels of the tree except the deepest are fully filled (complete binary tree)

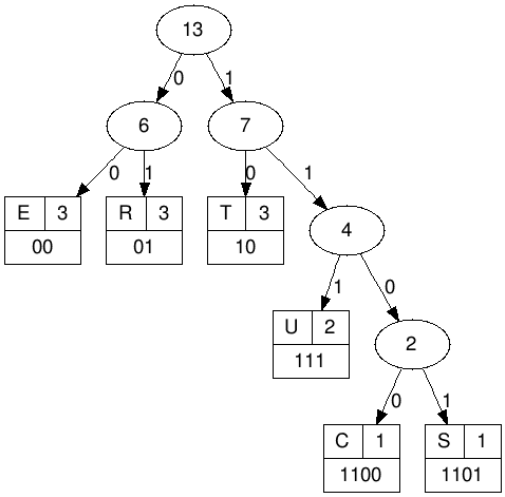
Compression, Huffman Coding

Compression consists in encoding data in fewer bits than its original representation. It is used when zipping files, to compress pictures, movies, etc…

Briefly summarise the purpose of Huffman coding.
Huffman coding is a lossless compression algorithm where:

- Every character is coded in a binary form.
- The most frequent characters have shorter codes and the less frequent characters have longer codes.

How can TREESTRUCTURE be represented in binary code?
How many bits are required in total?
In ASCII (using a byte=8bits for each character).



Creating a **huffman tree** is simple. Sort this list by frequency and make the two-lowest elements into leaves, creating a parent node with a frequency that is the sum of the two lower element's frequencies. You repeat until there is only one element left in the list. This element becomes the root of your binary huffman tree. To generate a huffman code you traverse the tree to the value you want, outputing a **0** every time you take a lefthand branch, and a **1** every time you take a righthand branch.

- In theory, Huffman coding is an optimal coding method whenever the true frequencies are known
- A Huffman code is not necessarily unique
- In a Huffman coding, the encoded codewords have different lengths -> hard to decode
- A fixed-length coding scheme is the most space efficient method if all letters have the same frequency.

Hashing

Hashing is a way to obtain both efficient memory use and effective retrieval. The key-value of a record is used to calculate the address where the record is to be stored.
The requirements of a hash function:

- must calculate an address within the legal address space.
- addresses should be uniformly distributed in the legal address space.
- efficient to compute.

The hash function can result in the same address for different keys. The **collisions** can be dealt with open or closed address hashing.

Open address Hashing:
Collision are solved by probing (i.e., searching through alternate locations in the array) until either the target record is found, or an unused array slot is found.
Example: Linear probing – if the key value has given the address to an already occupied space the next address space is considered.

Closed Address Hashing:
Example: Chaining – every entry in the storage is a list. All records with the same address end up in the same linked list.

1)The first problem was about generating the i-th member of a sequence

using a iteration
using recursion.

2)about a tree structure and you had to implement either preorder, postorder or inorder printing.

```
sub P(TreeNode)
   If LeftPointer(TreeNode) != NULL Then
      P(TreeNode.LeftNode)
   Output(TreeNode.value)
   If RightPointer(TreeNode) != NULL Then
      P(TreeNode.RightNode)
end sub
```

Tree transverse

Then they give you an example tree and ask you in what order your program will reach each node.

3)given a code of a program that calls itself Given 5 different entries

How many times for each of the entries the program will be called

And what is the time complexity of the program3
the number of elementary operations needed for an algorithm to finish.
Elementary operations:

- addition
- multiplication
- comparison
- memory write, swapping

4)find bugs in a program

A stack using structures

Given a hint that most bugs are related to things being pointers when they shouldn't be and vice versa

Then asked to implement something so the program can work as a queue as well
save the first pointer of the linked list

5) give an example of a stable sorting algorithm
Insertion and bubble sort

What other characteristics do sorting algorithms have
Properties: (time) complexity, stability, memory usage, in-place, adaptive, online, ...