Week 1 – Data Structures and Algorithms

**Elegant program** has an algorithm that is:

1. Easy to understand, code, and debug
2. Efficiently using computer’s resources

**Asymptotic analysis**

* Evaluates the efficiency of an algorithm
* Gives a way to define the inherent difficulty of a problem

**Data structure** = (in the most general sense) any data representation and its associated operations. Even an integer or floating-point number stored on the computer can be viewed as a simple data structure.

A solution is said to be **efficient** if it solves the problem within the required resource constraints.

**Data-centred** view of program design process:

1. Analyse the problem to determine the basic operations that must be supported, such as inserting a data item into the data structure, deleting a data item for the data structure, and finding a specified data item.
2. Quantify the resource constraints for each operation
3. Select the data structure that best meets these requirements

Resource constraints on certain key operations normally drive the data structure selection process. **Issues relating to the relative importance** of these operations are addressed by the following questions:

1. Are all data items inserted into the data structure at the beginning, or are the insertions interspersed with other operations? Static applications (where the data are loaded at the beginning and never change) typically get by with simpler data structures to get an efficient implementation, while dynamic applications often require something more complicated.
2. Can data items be deleted? If so, this will probably make the implementation more complicated.
3. Are all data items processed in some well-defined order, or is search for specific data items allowed? “Random access” search generally requires more complex data structures.

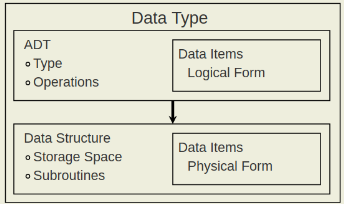
**Problem** = task to be performed it is best thought as a function in the mathematical sense (matching between inputs (the domain) and outputs (the range)). The values making up and input are function ***parameters***. A specific value for the parameters is called an ***instance*** of the problem.

**Algorithm** = method or a process followed to solve a problem. A given algorithm solves only one problem. Something can only be called an algorithm if it has the following properties:

1. It must be correct. It must compute the desired function, converting each input to the correct output.
2. It is composed of a series of concrete steps, meaning, the action described by that step is completely understood and doable by the person or machine that must perform the algorithm. Each stop must also be doable in a finite amount of time.
3. There can be no ambiguity as to which step will be performed next.
4. It must be composed of a finite number of steps.
5. It must terminate.

**Program** = instance or concrete representation of an algorithm. Terms algorithm and program are often used interchangeably but by definition an algorithm must provide sufficient detail that it can be converted into a program when needed.

**Type** = collection of values. Have both logical form and physical form. In terms of an ADT, the definition of the data type is its logical form, and the implementation is its physical form.



* **Simple type** = values contain no subparts, eg. an integer
* **Aggregate/composite type** = containing several pieces of information
* **Abstract data type (ADT)** = specification of a data type within some language, independent of an implementation. The interface for the ADT is defined in terms of a type and a set of operations on that type. The behaviour of each operation is determined by its inputs and outputs. An ADT does not specify how the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access (=**encapsulation**)

**Data item** = piece of information or a record whose value is drawn from a type. A member of a type.

**Data type** = type together with a collection of operations to manipulate the type. Integer is a member of the integer data type. Addition is an example of an operation on the integer data type.

**Array** = Contiguous block of memory locations, where each memory location stores one fixed-length data item. By this meaning it is a physical data structure. However, an array can also mean a logical data type composed of a (typically homogenous) collection of data items, with each data item identified by an index number.

**Data structure** = implementation for an ADT. In an object-oriented language, an ADT and its implementation together make up a **class**. Each operation associated with the ADT is implemented by a **member function** or a **method**. The variables that define the space required by a data item are referred to as **data members**. An **object** is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.

Data structure often refers to **data stored in a computer’s main memory.**

**File structure** = refers to the organization of data on peripheral storage, such as a disk drive or CD.

Week 2 – Comparing Algorithms

When estimating an algorithm’s performance, the **number of basic operations** required by the algorithm to process and input of a certain size is of primary consideration. Size is often the number of inputs processed. A basic operation must have the property that its time to complete does not depend on the particular values of its operands. Adding or comparing two integer variables are examples of basic operations but summing the contents of an array containing n integers is not, because the cost depends on the value on n (i.e., the size of the input).

Example

Text, letter

Description automatically generated

Size of the problem is A.length

The basic operation is to compare an integer’s value to that of the largest value seen so far.

It is reasonable to assume that it takes a fixed time to do one such comparison regardless of the value of the two integers or their position in the array.

Because most important factor affecting running time is normally size of the input, for a given input size **n** we often express **T** to run the algorithm as a function of n, written as **T(n)** always assuming the value is a non-negative.

**c** being the amount of time required to compare two integers in function largest, it does not matter to us what the precise value of **c** might be. The total time to run largest is therefore approximately **cn**, because we must make **n** comparisons, with each comparison costing **c** time. Expressed by the equation, the largest-value sequential search algorithm has a running time of **T(n)=cn.**

**Growth rate** = rate at which the cost of the algorithm grows as the size of its input grows.

Diagram

Description automatically generated with medium confidenceA picture containing chart

Description automatically generated

When analysing an algorithm normally we are not interested in the best case, because it might happen only rarely and generally be too optimistic for a fair characterization of the algorithm’s running time. If the best-case has a high chance of occurring, the **best-case analysis** is useful. The Quicksort algorithm can take advantage of the best-case running time of Insertion Sort to become more efficient.

The advantage of analysing **worst-case** is that you know for certain that the algorithm must perform at least that well. This is important for real-time applications.

**Average-case analysis** is not always possible to perform. It requires that we understand how the actual inputs to the program, and their costs, are distributed with respect to the set of all possible inputs to the program. For example, the sequential search algorithm on average examines half of the array values. This is only true if the element with value **K** is equally likely to appear in any position in the array. If this assumption is not correct, then the algorithm does not necessarily examine half of the array values in the average case.

* Incorrect assumptions about data distribution can have disastrous consequences on a program’s space or time performance. Unusual data distributions can also be used to advantage, such as is done by self-organizing lists.

Regardless of the algorithm’s growth rate: Constant factors never affect the relative improvement gained by a faster computer.

When you buy a faster computer or a faster compiler, the new problem size that can be run in a given amount of time for a given growth rate is larger by the same factor, regardless of the constant on the running-time equation.

For these reasons, **we usually ignore the constants** when we want an estimate of the growth rate for the running time or other resource requirements of an algorithm. This simplifies the analysis and keeps us thinking about **the most important aspect: the growth rate**. This is called **asymptotic** **algorithm** **analysis**. To be precise, asymptotic analysis refers to the study of an algorithm as the input size “gets big” or reaches a limit (in the calculus sense).

In some rare cases, where comparing algorithms meant to run on small values of **n**, the constant can have a large effect.

**Upper bound** = highest growth rate that the algorithm can have, **big-Oh notation**. If the upper bound for an algorithm’s growth rate (for, say, the worst case) is (f(n)), then we would write that this algorithm is “in the set O(f(n)) in the worst case”.

Table

Description automatically generated

Simplifying rules

Text

Description automatically generated

The first rule: if some function g(n) is an upper bound for cost function, then any upper bound for the function g(n) is also an upper bound for the cost function.

The second rule: any multiplicative constants can be ignored in equations when using big-Oh notation.

The third rule: given two parts of a program run in sequence, only the more expensive part needs to be considered.

The fourth rule: If some action is repeated some number of times, and each repetition has the same cost, then the total cost is the cost of the action multiplied by the number of times that the action takes place.

**Lower bound** = used to describe the least amount of a resource that an algorithm needs for some class of input. Denoted by Ω, pronounced “big-Omega”.

**Theta-notion** = When the upper and lower bounds are the same within a constant factor. Θ (big-Theta).

**Space/time tradeoff** **principle** = one can often achieve a reduction in time if one is willing to sacrifice space or vice versa. Many programs can be modified to reduce storage requirements by “packing” or encoding information. “Unpacking” or decoding the information requires additional time.

**Disk-based space/time tradeoff principle** = the smaller you can make your disk storage requirements, the faster your program will run. This is because the time to read information from disk is enormous compared to computation time, so almost any amount of additional computation needed to unpack the data is going to be less than the disk-reading time saved by reducing the storage requirements.

Instead of growth rate and running time, the term **computational complexity** is commonly used when describing how efficient the algorithm is.

Week 3 – OpenDSA: Lists

**List** = finite, ordered sequence of data items (elements). Close to mathematical concept of sequence.

* Ordered in this definition meaning that each element has a position in the list NOT that the elements are sorted by value.
* Each list element must have some data type
* Empty list contains no elements
* Number of currently stored elements is called the length of the list
* The beginning of the list is called **head** and the end of the list is called **tail**
* If there are ***n*** elements in a list the positions are given in a range from 0 – n-1

The two standard approaches to implementing lists are the **array-based list** and the **linked list**.

In the average case, insertion or removal of an element to or from a list requires moving half of the elements, which is Θ(n) .

Linked list uses dynamic memory allocation. Each element (or node) in a linked list contains the data stored in that element and in addition to this a **link (or a pointer )** to the next node (element).

When implementing a linked list structure many special cases must be taken in account and this makes the code more complicated and raises the changes for a bug to occur.

By implementing a **header node** as the first node of the list many special cases can be handled. This way special cases for empty list or when the current position is in the end of the list can be ignored. Rest of the special cases can be handled by adding a **trailer** node to the end of the list that never stores a value but acts as a place holder similarly to the header node.

**Overhead** = Additional space required by any container data structure such as a list to organize the elements being stored.

**Array based lists** have the disadvantage that their size must be predetermined before the array can be allocated. Array-based lists cannot grow beyond their predetermined size and when only a few elements are stored in the list, a substantial amount of space might be tied up in a mostly empty array. This empty space is the **overhead** required by the array-based list.

Array-based lists waste no space for an individual element whereas linked lists require the extra space for a pointer in each node. When the array for the array-based list is full, there is no wasted space (= no overhead), and this way it will be more space efficient by a constant factor than the linked list.

Linked lists only need the space for the objects on the list. There is no limit to the number of elements on a linked list, as long as there is free store memory available. The amount of space required by a linked list is Θ(n) while the space required by the array-based list is Ω(n),but can be greater.

Formula to determine which implementation is better for the case:

***n*** *= number of elements currently in the list*

***P*** *= size of a pointer in storage units (typically 4 bytes)*

***E*** *= size of a data element in storage units*

***D*** *= maximum number of list elements that can be stored in the array*

Amount of space required for the array-based list is **DE**, regardless of the number of elements actually stored in the list at any time.

Amount of space required for the linked list is **n(P+E)**

In general, the linked implementation required less space than the array-based implementation when relatively few elements are in the list. Conversely the array-based list becomes more space efficient when the array is close to full.

Using the equation, we can solve for n to determine the break-even point beyond which the array-based implementation is more space efficient in any particular situation. This occurs when:

n>DE/(P+E).

If  **P = E**, then break-even point is at **D/2.** That is, the array-based implementation would be more efficient (if the link field and the element field are the same size), whenever the array is more than half full.

As a rule of thumb: Linked lists are more space efficient when implementing lists whose number of elements varies widely or is unknown.

Time comparison

Array based lists are faster for access by position because positions can easily be adjusted forwards or backwards by the *next* and *prev* methods and they always take Θ(1) time.

Singly linked lists have no explicit access to the previous element, and access by position requires marching down the list from the front (or current position) to the specified position requiring Θ(n) time in both average, and worst cases.

Insert and remove in linked list require Θ(1) time, where as array-based lists must shift the remainder of the list up or down within the array requiring Θ(n) time. Often the time to insert/delete elements dominates all other operations and for this reason **linked lists are often preferred.**

Java and C++/STL vector classes implement dynamic array which allow the size to be changed after creating the array.

Stacks and Queues

**Stack**

* List-like structure in which elements may be inserted or removed from only one end
* Less flexible than lists
* Efficient and easy to implement
* Accessible element of the stack is called the **top** element
* Elements are not said to be inserted but **pushed** onto the stack
* When removed an element is said to be **popped** from the stack
* Can be implemented as array-based or linked similarly to lists
* LIFO

**Array-based and Linked Stacks**

* All operations in both implementations take constant time
* Array-based stack must declare a fixed-size array initially, and some of that space is wasted if the stack is not full.
* The linked stack can shrink and grow but requires the overhead of a link field for every element

**Queues**

* List like structure that provides restricted access to its elements
* Elements may only be inserted at the back (enqueue operation) and removed from the front (dequeue operation)
* Can be implemented as array-based or linked similarly to lists
* FIFO

**Array-based and linked queues**

* Array-based queues are tricky to implement efficiently
* Circular queues require us to keep count of elements in the queue or make the array be of size n+1 and only allow n elements to be stored in order to avoid pigeonhole principle case and thus being unable to distinguish full from empty queues.
* All member functions for both array-based and linked queue implementations require constant time. Space comparison issues are the same as for the equivalent stack implementations.

Searching an Array

**Sequential search** = moving through the list from beginning to the end searching for what you’re looking for.

If the sought-out item is found it is a **successful search** and if not, it is **unsuccessful search**.

In the worst case of sequential search an array that holds n values must be iterated through completely and the amount of work is proportional to n meaning the cost of the search is linear. For this reason, sequential search is sometimes called linear search.

Sequential search is the best that we can do when trying to find a value in an unsorted array but if the array is sorted in increasing order by value, then we can do much better with binary search.

**Binary search**

* Begins by examining the value in the middle position of the array; ***mid***
* If ***kmid=K*** then processing can stop immediately and in other cases the middle value provides useful information for finding the desired value of ***K*** as in sorted array we know if the value is before or after ***kmid***
* Cost on an array of n value is at most log(n)

Week 4 – Hashing

Hashing

* Method for storing and retrieving records from a database
* Insertion, deletion, and search for records based on a search key value
  + When properly implemented, these can be performed in **constant time**
* Stores records in an array called a hash table
* Hashing schemes place records in the table in whatever order satisfies the needs of the address calculation 🡪 **records are not stored by value**
* **Slot** = position in the hash table
* Most appropriate for answering the question “what record, if any, has key value K?”
* Search method of choice for application where all search is done by exact-match queries.
* Suitable for both in-memory and disk-based searching
* One of the most widely used methods for organizing large databases stored on disk

**Binning** = Type of hash function that divides and bins values to hash table slots based on the number of slots compared to the number of values.

**Mid-square method** = An approach to implementing a hash function. The key value is squared, and some number of bits from the middle of the resulting value are extracted as the hash code. When done correctly, the hash code will be affected by all bits of the key.

**Folding method** = An approach to implementing a hash function. Most typically used when the key is a string, the folding method breaks the string into pieces, converts the letter(s) to an integer value (typically by its underlying encoding value), and summing up the pieces.

The goal of a hash function is to minimize collision, but some collisions are unavoidable in practice. Thus, hashing implementations must include some form of collision resolution policy. These can be broken down into two classes: **open hashing (=separate chaining)** and **closed hashing (=open addressing)**. The difference between the two has to do with whether collisions are stored outside the table (**open hashing**), or whether collisions result in storing one of the records at another slot in the table (**close hashing**).

**Open hashing** is most appropriate when the hash table is kept **in main memory**, with the lists implemented by a standard in-memory linked list. One way to view open hashing is that each record is simply placed in a bin; while multiple records may hash to the same bin, this initial binning should still greatly reduce the number of records accessed by a search operation.

**Closed hashing** stores all records directly in the hash table where each record ***R*** with the key value ***kR*** has a home position that is **h(*kR)***. If ***R*** is to be inserted and another record already occupies its home position, then ***R*** will be stored at some other slot in the table. It is the business of the collision resolution policy to determine which slot that will be.

One implementation of closed hashing groups hash table slots into buckets. The ***M*** slots of the hash table are divided into ***B*** buckets, with each bucket consisting of ***M/B*** slots. The hash function assigns each record to the first slow within one of the buckets, if the slot is occupied, then the bucket slots are searched sequentially until an open slot is found. If a bucket is entirely full, then the record is stored in an overflow bucket of infinite capacity at the end of the table. All buckets share the same overflow bucket. A good implementation will use a hash function that distributes the records evenly among the buckets so that as few records as possible go into the overflow bucket.

Bucket methods are good for implementing hash tables stored **on disk** because the bucket size can be set to the size of a disk block. This way, when the entire bucket is in memory, processing an insert or search operation requires only one disk access, unless the bucket is full.

**Closed hashing with no bucketing, and a collision resolution policy that can potentially use any slot in the hash table** is the most used form of hashing.

* During insertion the goal of collision resolution is to find a free slot in the hash table when the home position for the record is already occupied this sequence of finding slots is called the **probe sequence** and it is generated by some **probe function**.
* **Linear probing** = The simplest approach to collision resolution is simply to move down the table from the home slot until a free slot is found.
* Linear probing has the tendency to cluster items together (=**primary clustering**). Small clusters tend to merge into big clusters and worsen the problem and leads to long probe sequences.

**Deletion from hash table**

* Two important considerations when deleting records from a hash table are:

1. It must not hinder later searches – the search process must still pass through the newly emptied slot to reach records whose probe sequence passed through this slot. Thus, the delete process cannot simply mark the slot as empty, because this will isolate records further down the probe sequence.
2. We do not want to make positions in the hash table unusable because of deletion. The freed slot should be available to a future insertion.

* Both problems can be resolved by placing a special mark in place of the deleted record called a **tombstone**.
* Tombstones allow searches to work correctly and allows reuse of deleted slots but a lot of tombstones lengthen the probing time. This can be resolved by for example periodically rehashing the table.

Week 5 – Recursion and Binary Trees

**Recursive** algorithm invokes itself to do part of its work.

**Recursion** (=process of solving a large problem by reducing it to sub-problem(s))

* Makes it possible to solve complex problems using concise, efficient, and easily understandable programs
* Recursive “call to itself” must be smaller than the originally attempted problem for a recursive approach to be successful
* In general, a recursive algorithm must have two parts
  1. The **base case** which handles a simple input that can be solved without resorting to a recursive call
  2. The **recursive part** which contains one or more recursive calls to the algorithm. In every recursive call, the parameters must be in some sense “closer” to the base case than those of the original call.
* Recursion is primarily as a tool for simplifying the design and description of an algorithm but because it involves function calls, it is usually more expensive than other alternatives such as while-loop.

Writing a recursive function:

* Basic four steps that are needed to write any recursive function
  + Write and define the prototype of the function
  + Write out a sample function call
  + Think of the smallest version of the problem (**base case**)
  + Think of smaller versions of the function call

Tracing recursive code

* When writing a recursive function, it should be thought in a top-down manner
  + Don’t worry about how the recursive call solves the sub-problem
  + Simply accept that it will solve the sub-problem correctly.
* When reading or tracing a recursive function, it must be considered how the function does its job

**Winding phase** = information (parameters) is passed from one recursive call to another, or even smaller problems, until a base case is reached.

**Unwinding phase** = a return value is passed back as the series of recursive calls unwinds.

Towers of Hanoi

* Ancient Vietnamese legend where a group of monks is tasked with moving a tower of 64 disks of different sizes according to certain rules and when they finish, the world will end.
* The puzzle begins with three poles and ***n*** rings, where all rings start on the leftmost pole (A). Each ring has different size and are stacked in order of decreasing size from the bottom. The problem is to move the rings from the leftmost pole to the middle pole (B) in a series of steps. At each step the top ring on some pole is moved to another pole. A ring may never be moved on top of a smaller ring.
* Assuming that a function X is available to solve the problem of moving the top ***n-1*** rings from pole A to pole C. Then move the bottom ring from pole A to pole B. Finally, again use function X to move the remaining ***n-1*** rings from Pole C to pole B. In both cases, X, is simply the Towers of Hanoi function called on a smaller version of the problem. Just moving 4 disks requires 15 moves. If each move took one second, it would take around 585 billion years to move all 64 disks because the required moves are at minimum 264-1.

Binary trees

Tree structures enable efficient access and efficient update to large collections of data.

Binary trees are widely used and relatively easy to implement.

Binary tree

* Made up of a finite set of elements called **nodes.** This set is either empty or consists of a node called **the root** together with two binary trees, called **the left and right subtree**, which are **disjoint** from each other and from the root (=no nodes in common). The roots of these subtrees are **children** of the root. There is an **edge** from a node to each of its children, and a node is said to be the **parent** of its children.
* If there is a path from node ***R*** to node ***M***,then ***R*** is an **ancestor** of ***M***, and ***M*** is a **descendant** of ***R***. Thus all nodes in a tree are descendants of the root and root is the ancestor of every node.
* The **depth** of a node ***M*** in the tree is the length of the path from the root of the tree to ***M***.
* The **height** of a tree is the depth of the deepest node in the tree. All nodes of depth dare at **level** d in the tree. The root is the only node at level 0, and its depth is 0.
* A **leaf node** is any node that has two empty children.
* An **internal node** is any node that has at least one non-empty child.
* ! All binary tree nodes have two children, one or both of which might be empty).
* **Full binary tree** is either an internal node with exactly two non-empty children or a leaf.
* **Complete binary tree** has a restricted shape obtained by starting at the root and filling the tree by levels from left to right. In a complete binary tree of height d, all levels except possibly the level d are completely full (the bottom level has its nodes filled in from the left side.)
* Any process of visiting all the nodes in some order is called a **traversal**.
* Any traversal that lists every node in the tree exactly once is called **an enumeration** of the tree’s nodes.
* **Preorder traversal** = visiting any given node before visiting its children.
* **Postorder traversal** = visiting each node only after visiting its children.
* **Inorder traversal** = first visit the left child (including its entire subtree), then visit the node, and finally visit the right child (and its entire subtree). In **binary tree search** this is used to print all nodes in ascending order of value.

**Recursive data structure** = data structure that is partially composed of smaller or simpler instances of the same data structure. E.g., linked lists and binary trees.

Binary Search Trees

**Binary search tree (BST)** is a binary tree that conform to the following condition, known as the **binary search tree property**. All nodes stored in the left subtree of a node whose key value is ***K*** have key values less than or equal to ***K***.

The shape of BST depends on the order in which elements are inserted. If a BST containing ***n*** nodes has the height ***n*** it means all elements were inserted in sorted order. In general, it is preferable for a BST to be **as shallow as possible** keeping the average cost of a BST operation low.

The cost for finding, inserting, or removing a value to/from binary tree in the worst case is the depth of the deepest node in the tree. Therefore it is desirable to keep the tree **balanced**. With a balanced tree, then the height of **n** node is approximately **log n**.

Data Structures and their qualities

**Hash Table**

* Allows for extremely fast exact-match search
* Records can be modified quickly when the modification does not affect its space requirements
* Supports efficient insertion of new records
* Deletion can be supported efficiently but too many deletions lead to some degradation in performance for the remaining operations
* Can be reorganized periodically to restore peak efficiency
* Can not perform efficient range queries

**Binary search tree**

* Supports large databases
* Supports efficient insertion and deletion of data records
* Supports range queries