**Introduction and Pseudocode**

Algorithms

An algorithm is a method or process followed to solve a problem

Algorithms must be correct, composed of a finite number of unambiguous steps and must terminate.

Data Structures

A data structure is a particular way of storing and organising data in a computer so it can be used efficiently.

The choice of data structure or algorithm can make a difference between a program running in a few seconds or days.

RAM (Random Access Machine) model of computation

-Memory consists of an infinite array

-Instructions are executed sequentially one at a time

-All instructions take unit time. Running time is the number of instructions executed.

For loops

For loops can iterate through a given set, not just consecutive integers

e.g. for value in {v1, v2, ….} do

Also can increment by a stated value

e.g. for i = 0 to 9; i+=2 do

**Arrays and Lists**

An array is a sequence of elements stored in consecutive locations in memory (contiguous), and all array elements are of the same data type (homogeneous).

The size of an array is fixed when it is declared.

To fetch the ith element in an array, search memory location k +(i-1)w where k = memory location of 1st element and w = width of a cell in memory

If an element is to be erased all subsequent elements have to be moved back one to preserve contiguity.

Linked lists are made up of nodes, each of which stores an element and a pointer or link to the next node. The first node is called the head, and the last node (tail) points to null. Nodes may be scattered all over memory, unlike arrays.

Linked lists contain references to the head node, tail node and size (int)

Implementation: L.head, L.tail and L.size are used to access the head and tail nodes and size.

N.data and N.next retrieve the data or next node for a given node.

e.g L.head.next.data gets the data stored in the second node

When inserting or removing from a linked list, references to predecessors and successors must be repaired

e.g to remove the head, set L.head = L.head.next and L.size -= 1, after checking if the list is empty. If the list only has one item and you are deleting it set L.head = NULL and L.tail = NULL instead. (see lecture slides for full pseudocode for deletion of head, and insertion)

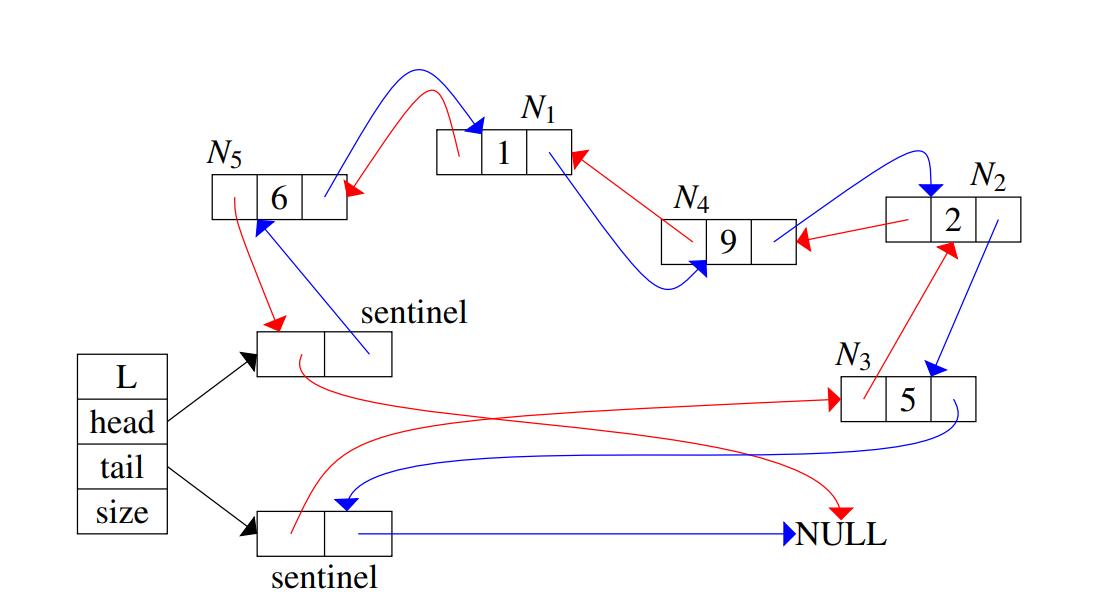
e.g to insert at the tail, set new element’s next pointer to NULL and set old tail’s next pointer to location of new element. Change the tail reference of the list to the location of the new tail.

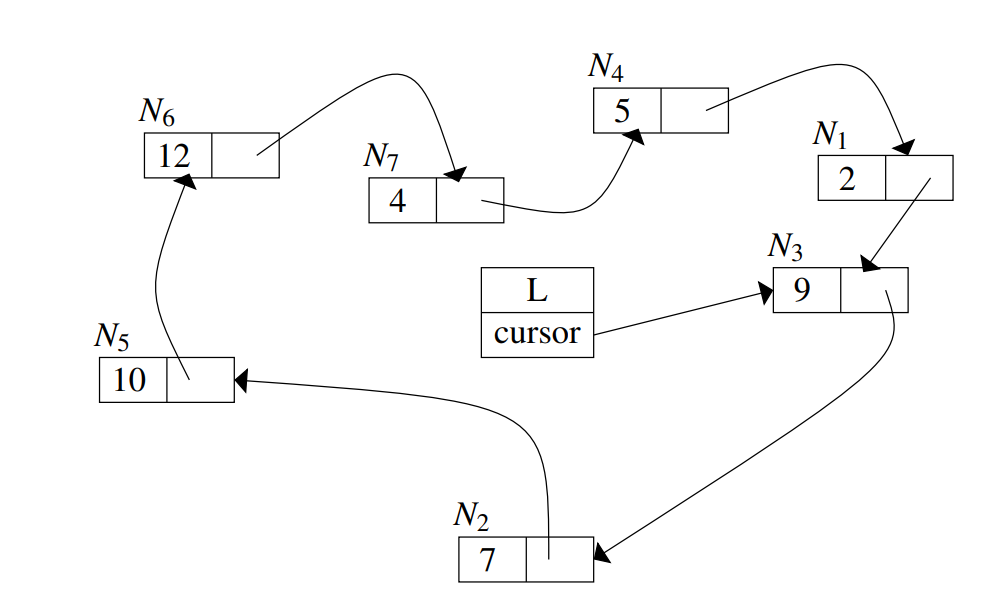
(pseudocode in slides)

Array has fast data access – can jump to ith element in array using array[i]. Linked list does not, since have to go through every preceding element first.

Insertion and deletion in arrays is slower as contiguity needs to be maintained, whereas it is faster with linked lists since only pointers for preceding element and the new element need to be updated.

Doubly linked lists store pointers to both the previous and next node in the list. 2 sentinel (dummy) nodes at head and tail - The prev node in the head points to NULL, as does the next pointer in the tail. An empty list only contains two sentinel nodes pointing to each other.



Circularly linked lists only contain next pointers, as with regular linked lists. Instead of the tail pointing to NULL, it points back to the first node, therefore there is no head or tail. Instead, a node is designated as the cursor, used as a starting point when traversing.

**Stacks and Queues**

A stack is a collection of objects that are inserted and removed according to LIFO. Objects can be inserted at any time,b but only the most recently inserted object can be removed.

Push(e) – inserts element e onto the top of the stack

Pop() – remove and return the top element of the stack, error occurs if stack empty.

Size – returns number of elements

Top – returns the top element without removing

isEmpty – checks if empty

In an array-based implementation, a stack consists of a N-element array S and an integer variable t that gives the top element. We initialise t = -1 and use this value of t to identify an empty stack.

Pseudocode for methods of array-backed stack in lecture slides

Array backed stacks are time efficient, however are limited by the fixed capacity of the array, meaning we waste memory when the size of the stack is less than the size of the array, but can cause exceptions if the stack grows too large. To fix this problem, a stack can be implemented using linked lists.

A stack can be implemented as a linked list, with the head of the list being the top of the stack.

A queue is a collection of items that are inserted and removed according to FIFO. Element access and deletion are restricted to the first element – the front of the queue. Element insertion is restricrted to the last element or the rear of the queue.

Methods:

Enqueue(e) – inserts element e at the rear of the queue

Dequeue() – removes and returns the element at the front of the queue, an error occurs iof the queue is empty

Size – number of elements in the queue

isEmpty – Boolean whether queue is empty

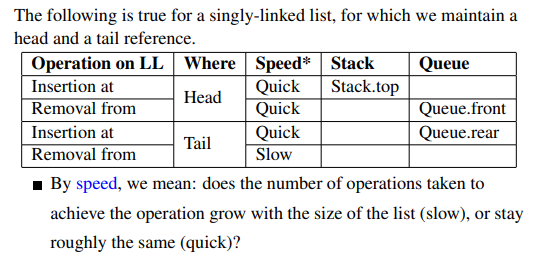
front – returns front element without removing it, error occurs if queue empty.

An array based implementation uses an array of a fixed size, and could have the first element of the array being the front, with enqueued elements are added to subsequent elements in the array. However, this is inefficient since all elements will have to be moved whenever the element is dequeued. Instead, pointers f and r are used: f points to the front of the queue while r points to the next available index in the array which is available (i.e. the empty cell behind the rear element). F and are initially both set to 0. F is incremented after every dequeue, r is incremented after ever enqueue.

For an array of size N, after N enqueue operations we will get an out of bounds error since arrays have a fixed size. To avoid this issue, let r and f wrap around the end of Q back to the start using modulo N arithmetic.

If the size of the queue is N then f = r and IsEmpty returns true even if its not empty. We avoid this by keeping the maximum elements that can be stored to N-1

The array based implementation of a queue is time efficient – all methods run in constant time, though the capacity of the queue is fixed. A linked list implementation can be used instead.



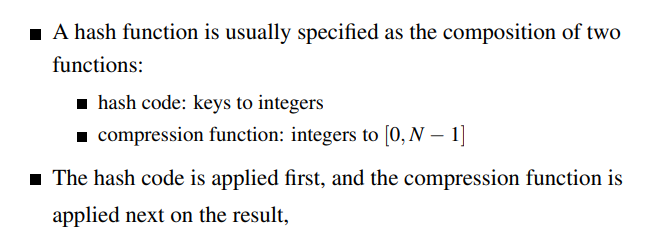
**Hash Tables**

If we want to store a large amount of data consisting of pairs of keys and values, you would want to look up the values using the keys, and perform these lookups many times on a large volume of data.

A hash table consists of a bucket array A of size N, where each cell is a “bucket” storing a collection of key-value pairs. The size N is the capacity of the hash table.

The hash function h is a function that maps each key k to an integer in range [0, N-1] where N is the capacity. h(k) is the index into the bucket array A (i.e. (k,v) is stored in position A[h(k)] in the bucket array.

Collisions occur when two key-value pairs end up in the same bucket. The hash function and the length of the backing array can be changed to try and reduce collisions.

There cannot be entries in a hash table with the same key, but can have the same value.

The goal of the hash function is to disperse the keys in as random and as uniform a way as possible.

The hash function should be efficiently computable, with each table position equally likely for each key. Compression functions include using division, or using multiply add and divide (MAD) where y maps to (ay + b) mod N where a, b >= 0

A large number of collisions reduces the performance of the hash table. A good hash function minimises collisions.

To lookup an element, perform the hash function on the key of the element you are looking up, and look at the index in the bucket array returned by this hash function.

Methods to remove collisions: (see topic 4 examples on blackboard for examples of each)

* Separate chaining – each bucket A[i] stores a list of the entries (k,v) such that h(k) = I (i.e. stores a list of all key-value pairs which map to that index in the bucket array).

Cost is proportional to length of list.

Average -> N/M where N is amount of data and M is size of array).

Worst case -> all keys hash to same list ☹

The remaining methods are open addressing schemes, meaning each bucket can only store a maximum of 1 entry.

* Linear Probing – If we try to insert an entry (k, v) into a bucket A[i] that is already occuipied, where i = h(k), then we try the next bucket, i.e. in A[(i+1) mod N]. If that position is also occupied, repeat the process until an empty bucket is found.

Insertion and search costs are dependent on length of the cluster of data

Average cluster length is N/M

Worst case = all keys hash to the same cluster

* Quadratic probing – Iteratively tries the buckets A[(i+f(j) mod N] for j = 0,1,2,… where f(j) = j2 until finding an empty bucket

Quadratic probing avoids clustering patterns that arise when using linear probing, however it may not find an empty bucket in the array A, even if there is one.

* Double hashing involves choosing a second hash function h’ , and if h maps a key k to a bucket A[i], with i = h(k), which is occupied, then iteratively try the buckets A[(i + f(j)) mod N], for j = 0,1,2,… and f(j) = j \* h’(k).

A common example of a secondary hash function is h’(k) = q – (k mod q), where q is a prime number < N. The secondary hash function should never evaluate to zero as this means it will check the same bucket again (which is already found to be occupied), since i+f(j) = i if h’(j) = 0

Comparison:

* Open addressing schemes are more memory efficient than separate chaining as they only use the fixed-size array and don’t introduce linked lists. They also make use of every element in the array rather than building up long linked lists in a small number of array elements while most array elements are empty.
* Separate chaining is usually competitive or faster than the other methods
* If memory space is not a major issue, the favourable method of collision handling is separate chaining.

Deletions:

* Deletion from a hash table must not hinder future searches – if a bucket is left empty future probes will see that it is empty and will not check subsequent buckets for collisions as it doesn’t know if the empty bucket means the data is not in the hash table at all, or if a collision occurred in the past but the data causing the collision was deleted at some point. Buckets should, however, not be left unusable.
* To solve this issue, markers called tombstones are left behind after a deletion. If a tombstone is encountered when searching, the algorithm will know to continue probing in order to account for past collisions. If a tombstone is encountered when inserting, probing should continue to check for duplicates, but then the new element should be placed in the bucket where the tombstone was found.
* The use of tombstones lengthenes the average probe sequence distance. This can be remedied by periodically rehashing all data items into a new hash table, or by reorganising records every time one is deleted to move records into the vacated bucket (or leave it empty if there are no other elements in the probe sequence)

**Recursion**

A recursive algorithm is one which calls itself as part of its execution. It must have a base case, must change its state such that it eventually reaches the base case, and must call itself recursively.

Memoisation – storing the results of intermediate function calls, when these results do not change over time, e.g. for Fibonacci function fib(n) calls fib(2), fib(3) etc multiple times during its execution. The results of these repeated calculations can be stored in a data structure such as a hash table to prevent the repeated calls to the same function to save execution time.

Backtracking – a technique for problems with many possible candidate solutions, but too many to try individually (e.g. sudoku has 6,670,903,752,021,072,936,960 possible combinations so don’t iterate through all of them)

General idea: build up a solution, if unable to continue then backtrack and make a change.

