Part 1

1 Introduction and Pseudocode

Algorithm - A method or process followed to solve a problem.
An algorithm must have:

- Correctness
- Composed of concrete unambiguous steps
- The number of steps must be finite
- Must terminate

Data structure - A particular way of storing and organising data in a computer so that it can be used efficiently

1.1 Machine Model

Random access machine:

- Memory consists of an infinite array
- Instructions executed sequentially one at a time
- All instructions take unit time. Running time is the number of instructions executed

2 Arrays and Lists

2.1 Linked list

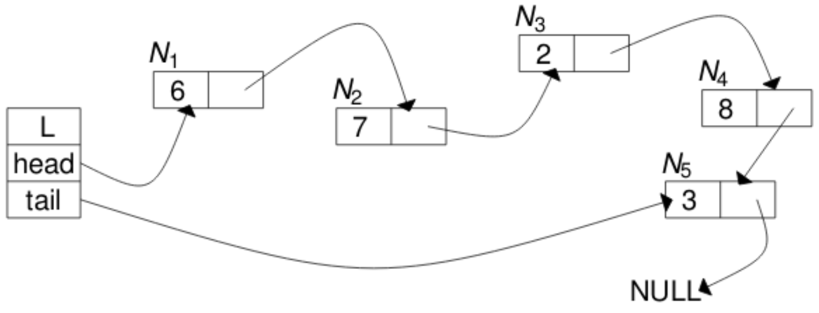
- A list is made up of nodes. Each node stores an element plus a pointer or "link" to another node
- The first node is called the head
- The last node, called the **tail** points to null
- The nodes may be scattered all over the memory

To implement this list we have pointers to the first and last node:

- `L.head`
- `L.tail`

May refer to the node N using

- `N.data`, the element
- `N.next`, the link, the next node in the list (may be `NULL`)



2.2 Doubly Linked Lists

A node in a doubly linked list stores two references

- a next link, which points to the next node in the list
- a prev link, which points to the previous node in the list

To simplify, we add two dummy or sentinel nodes at the ends of the doubly linked list

2.3 Circularly linked lists

In a circularly linked list the last element points to the first element rather than to NULL

3 Stacks and queues

3.1 Stacks

Stack - A collection of elements that are inserted and removed according to the last-in-first-out (LIFO) principle

A stack has the following methods:

- `push(e)`: insert element e at the top of the stack
- `pop`: remove and return the top element of the stack; an error occurs if the stack is empty

It may have the following methods:

- `size`: return the number of elements in the stack
- `isEmpty`: return a boolean indicating if the stack is empty
- `top`: return the top element in the stack, without removing it; an error occurs if the stack is empty

A stack can be implemented using an array as follows:

- the stack consists of an N element array S, an an integer variable t that gives the top element of the stack
- We initialise t to -1, and we using this value for t to identify an empty stack. The size of the stack is t+1

3.2 Queues

Queue - A collection of objects that are inserted and removed according to the first-in-first-out (FIFO) principle

A queue has the following methods:

- `enqueue(e)`: Insert element e at the rear of the queue
- `dequeue`: Remove and return from the queue the element at the front; an error occurs if the queue is empty

It may have the following methods:

- `size`: Return the number of elements in the queue
- `isEmpty`: Return a boolean indicating if the queue is empty
- `front`: Return the front element of the queue, without removing it; an error occurs if the queue is empty

A queue can be implemented using an array as follows:

- Use two variables f and r:
  - f is an index to the cell of Q storing the front of the queue, unless the queue is empty, in which case f=r
  - r is an index to the next available array cell in Q, that is, the cell after the rear of Q, if queue is not empty
- Initially f=r=0
- After each enqueue we increment r. After each dequeue we increment f

This is the issue that r keeps getting incremented, so we would get an array out of bounds error, to solve this use modulo N arithmetic on r and f to let them wrap around

4 Hash Tables

Hash table - Consists of a bucket array and a hash function
Bucket Array- An array A of size N where each cell A is though of as a bucket storing a collection of key-value pairs
Capacity - The size of the hash table

- The hash function is a function mapping each key to an integer in the range [0,N-1], where N is the capacity of the hash table
- The main idea is to use h(k) as an index into the bucket array A. That is, we store the key value pair (k,v) in the bucket A[h(k)]
- If there are two different keys with the same hash value h(k), then two different entries will be mapped to the same bucket in A. In this case, we say that a collision has occurred

A hash function is usually specified as the composition of two hash functions:

- Hash code: keys to integers
- Compression function: integers to [0,N-1]

The hash code is applied first, and the compression function is applied next on the result

The ideal goal is for a hash function to scramble the keys uniformly, but more practically:

- Hash function should be efficiently computable
- Each table position is equally likely for each key

Some compression functions:

- Division: take integer mod N
- multiply add and divide: y maps to ay+b mod N

4.1 Collisions

A large number of collisions reduces the performance of the hash table
A good hash function minimises the collisions as much as possible

Open addressing schemes - Store at most one entry in each bucket

4.2 Separate Chaining

Each bucket A[i] stores a list holding the entries (k,v) such that h(k)=i

Separate chaining performance:

- Cost is proportional to length
- Average length=N/M (N is amount of data, M is size of array)
- Worst case: all keys hash to the same list

We consider instead the expected worst case performance

- For any hash table consider the longest list in any bucket
- Then the expected worst case performance is the average length of this list

Assuming truly random hash functions the expected worst case cost of a lookup in a hash table with separate chaining is $\Theta(\log n / \log \log n)$

4.3 Second-Choice Hashing

- This is a refinement of separate chaining. Now we have two hash functions h_1 and h_2
- For a key k, compute $h_1(k)$ and $h_2(k)$ and store the key-value pair (k,v) in whichever bucket contains fewer items

Assuming truly random hash functions the expected worst-case cost of a lookup in a hash table with second choice hashing is $\Theta(\log \log n)$

4.4 Linear probing

- Try to insert into A[i], then A[(i+1) mod N], then A[(i+2) mod N] and so on until we find an empty bucket
- Insert and search costs depend on the length of the cluster
- Average length of cluster is N/M
- Worst case: All keys hash to the same cluster

Assuming truly random hash functions the expected worst-case cost of a lookup in a hash table with linear probing is $\Omega(\log n)$

4.5 Robin Hood Hashing

This is a variation of linear probing. If, during probing with a new key, an existing key is found that is "closer to home" than the new key, then the existing key is displaced and replaced by the new key

Assuming truly random hash functions the expected worst case lookup in a hash table with Robin Hood hashing is $\Omega(\log n)$

Assuming truly random hash functions the variance of the expected number of probes required in Robin Hood hashing is $O(\log \log n)$

4.6 Quadratic Probing

This iteratively tries the buckets

$$A[(i + f(j)) \bmod N], \text{ for } j = 0, 1, 2, \ldots, \text{ where } f(j) = j^2$$

Until finding an empty bucket

This avoids clustering patterns that occur with linear probing. However, it creates its own kind of clustering, called secondary clustering
This may not find an empty bucket in A even if it exists

4.7 Double Hashing

In this approach we choose a secondary hash function h', and if h maps some key k to a bucket A[i], with i=h(k) that is already occupied, then we iteratively try the buckets

$$A[(i + f(j)) \bmod N], \text{ for } j = 0, 1, 2, \ldots, \text{ where } f(j) = j\dot{h}'(k)$$

Common choice of secondary hash function is:

$$h'(k) = q - (k \bmod q), \text{ for some prime number } q < N$$

The secondary hash function should not evaluate to zero

4.8 Cuckoo Hashing

- With cuckoo hashing there are two tables, T_1 and T_2, and two corresponding hash functions h_1 and h_2
- Each key k is stored in either $T_1[h_1(k)]$ or $T_2[h_2(k)]$
- To lookup or delete a key can be done in time $O(1)$ since only two locations need be checked

Insertions:

- A key is stored in $T_1[h_1(k)]$ if it is empty
- Else it is stored in $T_2[h_2(k)]$ if that is empty
- If both $T_1[h_1(k)]$ and $T_2[h_2(k)]$ are occupies remove the key k' that is in $t_1[h_1(k)]$ and
  - Put k in $T_1[h_1(k)]$, and
  - move k' to $T_2[h_2(k')]$
  - If $T_2[h_2(k')]$ is occupied by another key k" them move that to $T_1[h+1(k'')]$
  - And so on

4.8.1 The cuckoo graph

- The cuckoo graph is a bipartite graph derived from a cuckoo hash table
- The vertices are the buckets, and for each key k, there is an edge from $T_1[h_1(k)]$ to $T_2[h_2(k)]$ (so we can think of keys as being the edges of the graph)
- An insertion into the table can be described as a path through the cuckoo graph

  The insertion of key k fails if the connected component of the cuckoo graph containing k contains two or more cycles

The insertion of key k succeeds if the connected component of the cuckoo graph containing k contains one cycle or no cycles

4.8.2 Cuckoo Hashing

- Cuckoo hashing fails if any connected component of the cuckoo graph has more than one cycle
- If we ensure the table are large enough this is very unlikely - having $M \geqslant 2N$ is sufficient

4.9 Comparison

- The open addressing schemes are more memory efficient compared to separate chaining
- Regarding running times, in experimental and theoretical analyses, the separate chaining method is either competitive or faster than the other methods
- If memory space is not a major issue, the best method is separate chaining

4.10 Deletions

- Deletions from the hash table must not hinder future searches. If a bucket is simply left empty this will hinder future problems
- But the bucket should not be left unusable
- To solve this problem we use tombstones: a marker that is left in a bucket after a deletion
- If a tombstone is encountered when searching along a probe sequence to find a key, we know to continue
- If a tombstone is encountered during insertion, then we should continue probing (to avoid creating duplicates), but then the new record can be places in the bucket where the tombstone was found
- The use of tombstones lengthens the average probe sequence distance

- Two possible remedies:
  - Local reorganization: after deleting a key, continue to follow the probe sequence of that key and move records into the vacated bucket (this will not work for all collision resolution policies)
  - Periodically rehash the table by reinserting all records into a new hash table. And if you have a record of which keys are accessed most these can be placed where they will be found most easily

5 Recursion

- A recursive algorithm must have a base case
- A recursive algorithm must change its state and move towards the base case
- A recursive algorithm must call itself, recursively

5.1 Backtracking

A technique for problems with many candidate solutions but too many to try
General idea: build up the solution one step at a time, backtracking when unable to continue

Generic algorithm

1. Do I have a solution yet?
2. No. Can I extend my solution by one ”step”?
3. If yes, do that
4. Do I have a solution now? If yes, I’m done
5. If not, try and extend again
6. When I can’t extend, take one step back and try another way
7. If no other extension available, then give up-no solution can be found