CS2040 Data Structures and Algorithms

AY23/24 S1

by Isaac Lai

Analysis of Algorithms

- Given a function $f(n)$, $g(n)$ is an asymptotic **upper bound** of $f(n)$, denoted $f(n) = O(g(n))$ if there exists a constant $c > 0$ and positive integer n_0 such that $f(n) \leq c \cdot g(n)\ \forall n \geq n_0$.
- Common growth terms: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$
- Ignore low-order terms and multiplicative constants in the higher-order term

Sorting

Selection sort

Given an array of n items,

1. Find the **largest** item.
2. **Swap** it with the item at the **end** of the array.
3. Go to step 1 by excluding the largest item from the array.

Time complexity: $O(n^2)$

Bubble sort

1. Move the largest item to the end of the array in each iteration by examining the i-th and $(i+1)$-th items
2. If their values are not in the correct order, i.e. $a[i] > a[i+1]$, **swap** them.

Time complexity: $O(n^2)$

Improvement For a sorted input, bubble sort is still $O(n^2)$. This can be improved by using a flag `isSorted`. Thus for sorted input, time complexity becomes $O(n)$.

Insertion sort

1. Start with the first element. Insert the next element into its proper sorted order.
2. Repeat previous step for the rest of the array.

Time complexity: $O(n^2)$. Best case of $O(n)$ when the array is already sorted.

Merge sort

Divide-and-conquer:

1. Divide the array into two (equal) halves.
2. Recursively merge sort the two halves.
3. Merge the two sorted halves to form a sorted array.

Time complexity: $O(n \log n)$

Quick sort

Also divide-and-conquer:

1. Choose a **pivot** item p and partition the array such that items in the first part are $< p$, and items in the second part are $\geq p$.
2. Recursively sort the 2 parts.

Worst case occurs when array is in decreasing order, and time complexity is $O(n^2)$. Best case occurs when the partition always splits the array into **2 equal halves**, in which case time complexity is $O(n \log n)$. In practice, the worst case is rare, so average time is $O(n \log n)$ (especially with randomised pivoting).

Radix sort

- Treats each element to be sorted as a character string.
- **Non-comparison** based sort
- In each iteration, organise the data into groups according to the **next** character

Time complexity: $O(n)$

Comparison of sorting algorithms

- A sorting algorithm is **in-place** if it only requires a constant amount of extra space.
- A sorting algorithm is **stable** if the relative order of elements with the same key value is preserved.

	Worst	Best	In-place	Stable
Selection	n^2	n^2	Yes	No
Insertion	n^2	n	Yes	Yes
Bubble	n^2	n^2	Yes	Yes
Improved bubble	n^2	n	Yes	Yes
Merge	$n \log n$	$n \log n$	No	Yes
Radix	n	n	No	Yes
Quick	n^2	$n \log n$	Yes	No

Lists

Array

Insertion at index i

1. Create a gap by shifting the i-th to last item rightwards by 1
2. Insert the new item in the gap
3. If the array is already filled, enlarge it by creating a new array (usually doubling the size of the original array) and copy the original array over. Insert the new item as per normal

Removal at index i: shift current items from index $i+1$ onwards to the left by 1

Other operations:

- `empty`, `size()`, `indexOf(int i)`(if not found return -1)
- `contains(int i)` – check if `indexOf(i)` returns -1
- (Retrieval) `getItemAtIndex(int i)`, `getFirst()`, `getLast()` (implement the last two using `getItemAtIndex`). $O(1)$ time due to array indexing
- (Insertion) `addAtIndex(int i, int item)` (use `insert`), `addFront(int item)`, `addBack(int item)` (use `addAtIndex`). Time complexity – best case: $O(1)$, worst case: $O(n)$, average case: $O(n)$ (due to enlargement step)
- (Deletion) `removeItemAtIndex(int i)` (use `remove`), `removeFront()`, `removeBack()` (use `removeItemAtIndex`). Time complexity – best case: $O(1)$, worst case: $O(n)$, average case: $O(n)$

$O(n)$ space complexity (best case is n and worst case is $2n$)

Linked List

- Each item in the list is stored as a **node**, which contains a **next pointer** that points to the node to its right
- The **head** node indicates the first node.

Accessing: have a reference that starts from the head, then iterate to move towrads the desired node

Insertion at index i

1. Create a new node `n`. Access the node `cur` at index $i-1$
2. Point next reference of `n` to neighbour of `cur`. Point next reference of `cur` to `n`
3. Increment number of nodes
4. If inserting at the front, point next of `n` to the head, then set the head to `n`

Removal at index i

1. Access the node `cur` at index $i-1$
2. Point next reference of `cur` to the neighbour of the i-th element
3. Decrement number of nodes
4. If the item to be removed is at the front, point the head to its next reference

Other operations:

- Retrieval – best case $O(1)$, worst case $O(n)$, average case $O(n)$
- Insertion – best case $O(1)$, worst case $O(n)$, average case $O(n)$
- Deletion – best case $O(1)$, worst case $O(n)$, average case $O(n)$

$O(n)$ space complexity

Other variants:

- Tailed Linked List – add a tail pointer in addition to head
- Circular Linked List – add a pointer from the tail node of the tailed linked list to the head node
- Doubly Linked List – add a previous pointer in addition to next

Array vs Linked List

- Only need to add/remove to front $\rightarrow$ LL
- Only need to add/remove to back $\rightarrow$ Array
- Need to add/remove anywhere in the list
  - Equal chance of adding at any index $\rightarrow$ no difference between LL or array
  - Need to keep adding/removing from a particular index $i \rightarrow$ use LL and maintain a reference to the node at index $i-1$
- Few insertions/deletions but a lot of accesses $\rightarrow$ Array

Stack and Queue

Stack

- Last-in-first-out (LIFO)
- Major operations – `push`, `pop`, `peek`
- Uses – calling a function, recursion, bracket matching, evaluating arithmetic expressions (postfix calculation), traversing a maze
- Array implementation – use a top index pointer
- Linked list implementation – top is the front of the list

Queue

- First-in-first-out (FIFO)
- Major operations – `poll` (dequeue), `offer` (enqueue), `peek`
- Array implementation
  1. Keep track of front and back indices
  2. Use a circular array, with $F = (F+1)\%\ maxsize$ and $B = (B+1)\%\ maxsize$
  3. Leave a gap to distinguish between full and empty arrays. Full: $(B+1)\%\ maxsize = F$, empty: $F = B$
- LL implementation – use tailed linked list

Hashing

Map

- Map is an ADT containing `<key, value>` pairs
- Duplicate keys are not allowed
- Basic operations (all $O(1)$ average time) – retrieval, insertion, deletion
- Mapping of a key to its value is done using a **hash function**

Direct Addressing Table

- Create an array where the index is the key
- Retrieval, insertion, deletion all done by indexing. For deleting, set relevant element to `null`
- Issues: keys must be nonnegative integers, range of keys must be small, keys must be dense

Hash Table

- Generalisation of direct addressing table
- Idea: hash function `h(key)` maps large integers to smaller ones, non-integer keys to integers
- Instead of indexing by `a[key]`, use `a[h(key)]`
- Problem: collision, i.e. two keys have the same hash value

Hash Functions

- Criteria for good hash functions:
  - **Consistent** same key maps to same buckets
  - **Fast** to compute
  - Distributes keys as **uniformly** as possible to the buckets. Keys are distributed to buckets with equal probability, and every bucket has some key hashing to it
- Perfect hash function: one-to-one mapping between keys and hash values. Possible if all keys are known beforehand
- Picking the hash table size is important. In general, it should be prime

Examples:

- Division method – for a hash table with m slots, use the modulo operator to map an integer to a value between 0 and $m-1$
- Multiplication method – multiply by a constant real number $0 \leq \alpha \leq 1$, extract the fractional part, multiply by m, i.e. $hash(k) = \lfloor m(k\alpha - \lfloor k\alpha \rfloor) \rfloor$. $\alpha = \frac{1}{\phi}$ is a good choice
- Strings – sum up ASCII values of all characters and "shift" the sum after each character

Collision Resolution

Analysis of hash table performance

- n – number of keys in the hash table
- m – size of the hash table − number of slots
- $\alpha = \frac{n}{m}$ – **load factor**, a measure of how full the hash table is

Criteria

- Minimise clustering
- Always find an empty slot if it exists
- Give different probe sequences when 2 initial probes are the same
- Fast

Separate Chaining

- Use a linked list to store collided keys
- `insert(key, data)`: add to the back of the list at `a[h[key)]`
- α is the average length of the linked lists
- To keep α bounded, we may need to reconstruct the whole table when the load factor exceeds the bound. When the bound is exceeded, we need to rehash all the keys into a bigger table

Linear Probing

- Finding and insertion: go to the next empty slot
- Deletion: **lazy** deletion, use three different states of a slot – occupied, deleted, empty
- Problems: **primary clustering** i.e. creating consecutive occupied slots, increasing the run time of the operations
- Modified linear probing: $(\texttt{hash(key)} + k * d)\%m$ where d is a constant integer > 1 and coprime to m

Quadratic Probing

- Probe sequence: $(\texttt{hash(key)} + k^2)\%m$
- If $\alpha < 0.5$ and m is prime, then we can always find an empty slot
- Problem: **secondary clustering** if two keys have the same initial position, their probe sequences are the same

Double Hashing

- Use a secondary hash function $\texttt{hash}_2$
- Probe sequence: $(\texttt{hash(key)} + k * \texttt{hash}_2\texttt{(key)})\%m$
- Secondary hash function must not evaluate to 0

For both separate chaining and open addressing, worst case: $O(n)$, best case: $O(1)$

Heap

- Motivation: priority queue
- **Complete** binary tree – every level except possibly the last is completely filled, and all nodes are as far left as possible
  - Height – $O(\log n)$
  - `parent(i) = floor(i/2)` execpt for `i = 1`
  - `left(i) = 2*i`, no left child when `left(i) > heapsize`
  - `right(i) = 2*i+1`, no right child when `right(i) > heapsize`
- Binary heap property (except root) – `A[parent(i)]` $\geq$ `A[i]` (Max heap)
- Largest element stored at the root
- All are $O(\log n)$
  - `Insert` – use `ShiftUp`
  - `ExtractMax` – use `ShiftDown`
- `CreateHeap` – $O(n)$
- Can do sorting in $O(n \log n)$ time by just calling `ExtractMax` n times

UFDS

- Collection of disjoint sets with each modelled as a tree
- Each set's representative item is its root
- Forest of trees modelled with an array `p` where `p[i]` is the parent of item `i`
- `p[i] = i` $\implies$ `i` is a root
- `findset(i)` – recursively visit `p[i]` until `p[i] = i` and compress the path along the way
- `isSameSet(i, j)` – check if `findSet(i) = findSet(j)`
- `unionSet(i, j)` – set representative item of the taller tree as the new representative item of the combined set
  - "Union-by-rank", helps keep the combined tree shorter
  - Use array `rank` where `rank[i]` stores the upper bound of height of subtree rooted at `i`
- All operations run in $O(\alpha(N))$ time with union-by-rank and path compression. $\alpha(N)$ is the inverse Ackermann function

BST

- BST property property – for all vertices `x` and `y`, `y.key < x.key` if `y` is in left subtree of `x`, `y.key` $\geq$ `x.key` if `y` is in right subtree of `x`
- `search`, `findMin()/findMax()`, `insert`, `successor/predecssor`, `delete` all run in $O(h)$
- Inorder traversal – $O(n)$
- Worst case: $h = O(n)$

Balanced BST (AVL Tree)

- Motivation – keep $h = O(\log n)$
- Height – number of edges on path from current vertex to deepest leaf, size – vertices of the subtree rooted at the vertex
- Balance factor – `x.left.height - x.right.height`. If absolute value ≤ 1, `x` is height-balanced
- `bf(x) = +2` and $0 \leq$ `bf(x.left)` ≤ 1 – `rightRotate(x)`
- `bf(x) = +2` and `bf(x.left) = -1` – `leftRotate(x.left)` then `rightRotate(x)`
- `bf(x) = -2` and $-1 \leq$ `bf(x.right)` ≤ 0 – `leftRotate(x)`
- `bf(x) = -2` and `bf(x.right) = 1` – `rightRotate(x.right)` then `leftRotate(x)`
- Insertion – insert as in normal BST, then walk up AVL tree from insertion point to root while checking if any vertex is out of balance. Can only trigger one of the four rebalancing cases once
- Deletion – delete as in normal BST, similar to insertion but the cases can be triggered more than once

Graphs

- Complete graph – simple graph with N vertices and ${}^{N}C_2$ edges
- In/out degree of a vertex – number of in/out edges from a vertex
- Component – a maximal group of vertices in an undirected graph that can visit each other via some path
- Connected graph – undirected graph with 1 component
- Tree – connected graph, $E = V - 1$
- Bipartite graph – undirected graph where we can partition the vertices into two sets so that there are no edges between members of the same set
- Adjacency matrix – space complexity $O(V^2)$
- Adjacency List – space complexity $O(V + E)$
- Edge list – space complexity $O(E)$ ($E = O(V^2)$)