CS2040S Data Structures and Algorithms

AY 24/25 Sem 2 — github/omgeta

1. Asymptotic Analysis

Preconditions are conditions which must be true for the algorithm to work correctly. Postconditions are conditions guaranteed to be true after the algorithm ends.

Invariants are conditions that remain unchanged and consistently true throughout the algorithm.
Loop invariants are invariants for each loop iteration.

Runtime $T(n)$ is given if $\exists c, n_0 > 0$ s.t. $\forall n > n_0$:

i. $T(n) \leq cf(n) \iff T(n) = O(f(n))$
ii. $T(n) \geq cf(n) \iff T(n) = \Omega(f(n))$
iii. $T(n) = O(f(n)), \Omega(f(n)) \iff T(n) = \Theta(f(n))$

Order of growth is:
$O(1) < O(\log\log n) < O(\log n) < O(\log^2 n) < O(\sqrt{n}) < O(n) < O(n\log n) < O(n^2) < O(2^n) < O(2^{2n}) < O(n!)$

Common recurrences:

i. $T(n) = T(n/2) + O(1) = O(\log n)$
ii. $T(n) = T(n/2) + O(n) = O(n)$
iii. $T(n) = 2T(n/2) + O(1) = O(n)$
iv. $T(n) = 2T(n/2) + O(n) = O(n\log n)$
v. $T(n) = T(n-1) + O(1) = O(n)$
vi. $T(n) = T(n-1) + O(n) = O(n^2)$
vii. $T(n) = 2T(n-1) + O(1) = O(2^n)$

Mathematical Properties

Useful properties:

i. $\log(xy) = \log x + \log y$ (Product rule)
ii. $\log(x/y) = \log x - \log y$ (Quotient rule)
iii. $\log(x^k) = k\log x$ (Power rule)
iv. $\log_b x = \frac{\log_a x}{\log_a b}$ (Change of base)
v. $x^a \cdot x^b = x^{a+b}$ (Product rule)
vi. $\frac{x^a}{x^b} = x^{a-b}$ (Quotient rule)
vii. $(x^a)^b = x^{ab}$ (Power rule)

Master Theorem

Dividing function $T(n) = aT(n/b) + f(n)$ where $a > 1, b > 1, f(n) = \Theta(n^k \log^p n)$ has cases:

i. $\log_b a > k \implies T(n) = \Theta(n^{\log_b a})$
ii. $\log_b a = k \wedge p > -1 \implies T(n) = \Theta(n^k \log^{p+1} n)$
$\wedge\, p = -1 \implies T(n) = \Theta(n^k \log\log n)$
$\wedge\, p < -1 \implies T(n) = \Theta(n^k)$
iii. $\log_b a < k \wedge p \geq 0 \implies T(n) = \Theta(n^k \log^p n)$
$\wedge\, p < 0 \implies T(n) = \Theta(n^k)$

Decreasing function $T(n) = aT(n-b) + f(n)$ where $a > 0, b > 0$ has cases:

i. $a < 1 \implies T(n) = O(f(n))$
ii. $a = 1 \implies T(n) = O(n \cdot f(n))$
iii. $a > 1 \implies T(n) = O(a^{n/b} \cdot f(n))$

2. Searching

Binary search, $O(\log n)$, in a sorted array cuts search space in half until the target is found or range exhausted.

```
int binarySearch(T[] arr, T x) {
  int lo = 0, hi = arr.length - 1;
  while (lo < hi) {
    int mid = lo + (hi - lo) / 2;
    if (arr[mid] < x) lo = mid + 1;
    else hi = mid;
  }
  return lo;
}
```

1D Peak Finding, $O(\log n)$:

i. Binary search on side with rising neighbour if not peak

2D Peak Finding on $n \times m$, $O(n \log m)$:

i. Take global max of column, binary search on columns with rising neighbour if not peak.
ii. Optimise: Quadrant DnC. $O(n+m)$.

kth Smallest, $O(n)$:

i. QuickSelect, recurse on half with k

3. Sorting

BubbleSort repeatedly swaps inverted adjacent elements up to form a globally sorted suffix.

i. After i iterations, last i elements are correct
ii. Time: $O(n)$ (Best), $O(n^2)$ (Worst)
iii. In-place and stable

SelectionSort selects the smallest unsorted element and places it in the globally sorted prefix.

i. After i iterations, first i elements are correct
ii. Time: $O(n^2)$ (Best & Worst)
iii. In-place but unstable

InsertionSort adds elements to a locally sorted prefix.

i. After i iterations, first i elements are sorted
ii. Time: $O(n)$ (Best), $O(n^2)$ (Worst)
iii. In-place and stable

MergeSort divides the array into halves, sorts each half, and merges the sorted halves.

i. Locally sorted prefix in power of 2
ii. Time: $O(n\log n)$ (Best & Worst)
iii. Space: $O(n\log n)$
iv. Out-of-place and stable

QuickSort partitions the array around $\leq$ and $>$ pivot, then recursing on both partitions.

i. Elements before pivot are $\leq$, after pivot are $>$
ii. Time: $O(n\log n)$ (Best), $O(n^2)$ (Worst)
iii. In-place but unstable
iv. Optimise: randomized pivot, 3-way partition for $O(n\log n)$ worst-case, insertion sort for small n

CountingSort counts number of objects with distinct keys.

i. Time: $O(n+k)$ where k is number of keys
ii. Space: $O(n+k)$

RadixSort sorts integers by individual digits.

i. Time: $O(nd)$ where d is number of digits
ii. Space: $O(n+k)$

4. Trees

Binary Search Trees (BSTs)

Binary search tree maintains that for any node, all left descendants are $\leq$ and all right descendants are $>$. Operations are $O(h)$ where h is the height of the tree.

BSTs are balanced if $h = O(\log n)$.
If balanced, (non-traversal) operations are $O(\log n)$. In the worst case, they are $O(n)$.

i. Height, h is given by $\log n - 1 \leq h \leq n$
ii. `search`: Traverse left/right based on comparisons.
iii. `insert`: Find position via search and add node.
iv. `delete`: If not leaf, replace with successor.
v. Traversal yields sorted elements. Time: $O(n)$

AVL Trees

Self-balancing BSTs which are height-balanced, i.e. height difference between left/right subtrees differ by at most 1.

i. Balance factor $= |height_{left} - height_{right}| \leq 1$.
Nodes are left-heavy with balance factor > 1, or right-heavy with balance factor < -1.
ii. $h < 2\log n \iff 2^{\frac{h}{2}} \leq n \leq 2^{h+1} - 1$
iii. `insert`: After insertion, walk up tree and fix lowest unbalanced node (max 2 rotations)
iv. `delete`: After deletion, fix all unbalanced nodes until root (up to $\Theta(\log n)$ rotations)

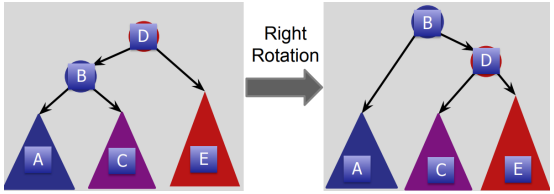
To balance left-heavy node `v`:

i. `v.left` balanced or left-heavy
$\implies$ `rightRotate(v)`
ii. `v.left` right-heavy
$\implies$ `leftRotate(v.left); rightRotate(v)`

To balance right-heavy node `v`:

i. `v.right` balanced or right-heavy
$\implies$ `leftRotate(v)`
ii. `v.right` left-heavy
$\implies$ `rightRotate(v.right); leftRotate(v)`

Rotations




```
Node leftRotate(B)          Node rightRotate(D)
  D = B.right                 B = D.left
  D.par = B.par               B.par = D.par
  B.par = D                   D.par = B
  B.right = D.left            D.left = B.right
  D.left = B                  B.right = D
  return D                    return B
```

Problems

Order Statistics, $O(\log n)$ if balanced

i. Augment nodes with subtree size $= size_{left} + size_{right} + 1$
ii. `select`: find kth smallest element.
iii. `rank`: get node position in sorted order.

Counting Inversions, $O(n \log n)$:

i. After inserting each element, add `i - tree.rank(ele)` to running count.

Interval Query, $O(\log n)$

i. Nodes store interval, key is lower interval bound
ii. Augment nodes with subtree max upper bound
iii. `interval`: If left subtree exists with $max \geq x.low$, recurse on left subtree, else on right subtree

1D Range Query, $O(\log n + k)$ for k elements in range:

i. Leaves are elements. Internal nodes are max of left subtree.
ii. `range`: Find split node $O(\log n)$ and traverse leafs

2D Range Query, $O(\log^2 n + k)$ for k elements in range:

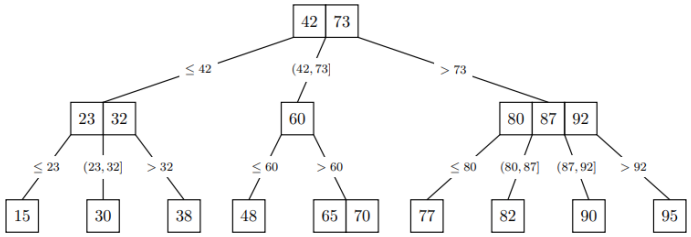
i. Build 1D x-tree for each x, then for each internal node, build y-tree for all y
ii. `range`: $O(\log n)$ y-tree searches each of $O(\log n)$

Tries

Tree storing strings where each node is a character, and each path is a key. Operations are $O(L)$, where L is length of the key. Faster than $O(Lh)$ of strings in bBST. Both use O(text size) space but Trie has node overhead.

i. `prefixSearch`: Find all keys with prefix.
Time: $O(L + k)$, where k matches

(a,b)-Trees



Self-balancing search trees where non-root nodes have between a, b children for $2 \leq a \leq \frac{b+1}{2}$

i. Root has $[2, b]$ children.
Internal nodes have $[a, b]$ children.
ii. Nodes with k children have $k - 1$ keys.
Leaves have $[a - 1, b - 1]$ keys.
iii. All leaves at same depth
iv. `search`: Time: $O(b \log_a n) = O(\log n)$
v. `insert`: Proactively split full nodes ($b - 1$ keys) during search, guaranteeing parent of split node won't have excess keys after insertion of split node.
Time: $O(\log n)$
vi. `split`: Split about median node z with $\geq 2a$ keys, s.t. left has $\geq a - 1$ keys and right has $\geq a$ keys. Then, z offered to parent with $\leq b - 2$ keys.
Time: $O(b)$ for splitting node with b children
vii. `delete`: Passively, delete node then recursively check parents for violation, carrying out merge/share with smallest adjacent sibling. For internal nodes, replace with successor.
Time: $O(\log n)$
viii. `merge`: Demote split node and join when $< b - 1$ keys. Time: $O(b)$
ix. `share`: `merge` then `split` when $\geq b - 1$ keys