CS2040S Midterm Summary

By Chen Yiyang, AY20/21 Sem 2

Order of Growth

O for upper bound, Ω for lower bound

$$T(n) = \Theta(f(n)) \iff T(n) = O(f(n)) = \Omega(f(n))$$

- i.e. upper and lower bound the same

Some common ones:

$T(n) = \log(n!) = O(n \log n)$

$T(\text{fib}(n)) = O(\phi^n)$, $\phi = 0.618...$ golden ratio

Master Theorem (extra)

$$T(n) = aT(\frac{n}{b}) + f(n), \quad a \geq 0, b > 1$$

$$= \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) < n^{\log_b a} \text{ polynomially} \\ \Theta(n^{\log_b a} \log n) & \text{if } f(n) = n^{\log_b a} \\ \Theta(f(n)) & \text{if } f(n) > n^{\log_b a} \text{ polynomially} \end{cases}$$

[Algo] Binary Search

Invariant:

[1] A[begin] <= key <= A[end]

[2] range <= n/2^k for iteration k

Criteria for using Binary Search

[1] function **monotonic**

[Algo] Peak Finding

```
FindPeak(A, n)
    if A[n/2] is a peak then
        return n/2
    else if A[n/2+1] > A[n/2] then
        search for peak in right half
    else if A[n/2-1] < A[n/2] then
        search for peak in left half
```

[Algo] Sorting

Bubble Sort

~ in-place, stable

~ Invariant: largest n items at the end correctly after n iterations

~ Time: best O(n), avg./worst O(n^2)

Selection Sort

~ in-place, unstable (cuz. swapping)

~ Invariant: smallest n items in front correctly after n iterations

~ Time: best/avg./worst O(n^2)

Insertion Sort

~ in-place, stable

~ Invariant: first n items sorted correctly after n iterations.

~ Time: best O(n), avg./worst O(n^2)

~ Good at sorting almost sorted array

Merge Sort

~ not in-place: O(nlogn) space, stable

~ Time: best/avg./worst O(nlogn)

Quick Sort

~ Time: avg. O(nlogn), worst O(n^2)

~ Improvement 1 - Probabilistic Quick Sort: repeat partitioning until an acceptably good pivot is found. Expectation follows that of a geometric r.v. -> Always O(nlogn)

~ Improvement 2 – Double Pivot -> 3 regions: smaller, equal, and larger than pivot

Quick Select

To find the k-th largest / smallest item. Similar to Quick Sort, but only need to recurse on one subarray every time.

~ Time: O(n), if probabilistic

[DS] AVL Tree

Balanced Tree: height of tree = O(logn)

AVL: height-balanced, i.e., for all nodes **v**

$$|v.\text{left.height} - v.\text{right.height}| \leq 1$$

Left/right Heavy: left/right child larger height

Successor

```
FUNCTION successor(node)
  IF node.right != null THEN
    return searchMin(node.right)
  ENDIF

  parent <- node.parent
  child <- node
  WHILE ((parent != null) && (child == parent.right))
    child <- parent
    parent <- child.parent
  ENDWHILE
  return parent
END
```

Rotate

left-/right-rotate == root goes to left/right

(Remember to update height/weight/...)

If `v` is out of balance and left-heavy:

- `v.left` is balanced or left-heavy: `rightRotate(v)`
- `v.left` is right-heavy: `leftRotate(v.left)` then `rightRotate(v)`

If `v` is out of balance and right-heavy:

- `v.right` is balanced or right-heavy: `leftRotate(v)`
- `v.right` is left-heavy: `rightRotate(v.right)` then `leftRotate(v)`

(personal intuition: treat the above 1 or 2 operations as 1 "rotate set", which reduces height difference by 1)

Insert

~ Time O(logn)

~ Rotations: 2 (i.e. "1 set") –> check upwards and adjust first unbalanced

Delete

First delete (using successor) then adjust

~ Time O(logn)

~ Rotations: O(logn) (i.e. "O(logn) sets")

-> check all the way up till root & adjust all unbalanced

delete(u) based on cases:

[1] no children: delete & adjust

[2] 1 child: link u's child to u's parent. Adjust

[3] 2 children: swap u with u's successor, then delete as in case [1] or [2]

[DS] Trie

Tree but not binary tree. Each node stores a char & a special char (for terminal of a string) -> Each root-to-leaf path represents a word

~ rationale: to minimize string comparisons

~ Time: O(L) for search/insert

Space: O(nL) = O(size of text * overhead)

L: max length, n: #strings

[DS] Augmented Tree

Augmented: modify ADT and add in extra data for certain features / functionality.

Order Statistics

Goal: find the k-th largest / smallest item in the tree (i.e. search by rank)

Extra Data: **weight** of each node

~ rank **in subtree** = left.weight+1

~ Time: O(logn) for insert/delete/query

Interval Query

Goal: find an interval / range that covers the current query point

Extra Data: **max endpoint** of the subtree rooted at current node

```
FUNCTION intervalSearch(x):
  c <- root
  WHILE (c not null AND x not in interval of c) DO
    IF (c.left null OR x > c.left.max) THEN
      c <- c.right
    ELSE
      c <- c.left
    ENDIF
  ENDWHILE
  return c.interval
END
```

~ Intuition: always go left unless it is obviously wrong to go left

~ Time: O(logn) for insert/delete/query

~ Limitations: 1) not find the smallest / most suitable interval 2) some (right) intervals will never be returned

Orthogonal Range Search (1D)

Goal: find all the points in a query interval (opposite of Interval Query)
Augmentation: each **leaf** node in a bBST stores data, and each **non-leaf** stores the **largest value in its left subtree**.
~ query(): find "split node" then do left and right traversal (both symmetric)

```
FUNCTION leftTraversal(v, low, high)
  // rightTraversal symmetric
  IF (low <= v.key) THEN
    all-leaf-traversal(v.right)
    leftTraversal(v.left, low, high)
  ELSE
    leftTraversal(v.right, low, high)
  ENDIF
END
```

~ Time: O(logn+k) for query, O(logn) for insert/delete
  n: #nodes in total, k: #nodes in range

[DS] Hashing

(notations: n #items, m #buckets)
Symbol table
~ do not support predecessor/successor
~ keys are **immutable** and no duplicates
Collision: distinct keys hashed to same bucket
Simple Uniform Hashing Assumption:
[1] Each key, equally likely to be mapped to each bucket (among all buckets)
[2] Each key is mapped independently
Double Hashing (f, g hash functions)

$$h(k, i) = (f(k) + i \times g(k)) \bmod m$$

Solution 1: Chaining

Every bucket is a linked list, not a var. (can insert even if n>m)
~ Time for insert O(1) (to front of LL)
~ Expected search time: 1+n/m = O(1)
~ Max. search time when n==m: O(logn)

Solution 2: Open Addressing

Probe a seq. of buckets until an empty one is found and place the item there. (cannot insert once n==m)
~ Linear probing: probe linear seq.
~ Implementation
  ~ insert: probe until an empty or DELETED bucket and insert item there
  ~ search: probe until found or not found if probe one empty or all m probed.
  ~ delete: search for the item and mark the bucket as DELETED (not empty!!!)
  ~ Grow table size: m -> 2*m when full
~ Time: <= 1/(1-alpha) for all operations
  alpha = n/m, **load** of hash table

Recitations

Rec. 03 – Fisher-Yates/Knuth Shuffle

```
PROCEDURE KnuthShuffle(A[1..n])
  FOR i <- 2 TO n
    r <- random(1, i)
    swap(A, i, r)
  ENDFOR
END
```

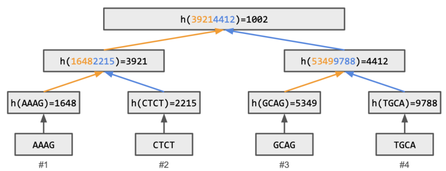
Rec. 04 – (a,b) Tree & B-Tree

B-Tree == (B, 2B) Tree
Definition of (a, b)-Tree
[1] Each non-leaf min. a (2 for root), max. b children.
[1.1] #children == #key+1 for non-leaf; min. (a-1), max (b-1) keys for leaf
[2] Key-ordering for children.
[3] All leaf nodes same height (growing upwards)
Basic Operations:
~ split: give 1 key to parent, and then split this node & its remaining keys to 2 new nodes.
~ merge: pull 1 key down from parent, and merge the two keys divided by this key into one new child node.
~ share: share == merge + split
Other Operations:
~ insert: insert, then check up to root, split if necessary
~ delete: delete, then check up to root, merge/share if necessary.
Analysis:
~ Height min O(log_b n), max O(log_b n)
~ Time: O(logn) for insert/delete/search
~ same order as normal bBST, but smaller coefficient -> shorter tree and better for caching / mem. Access.

Rec. 05 – Merkle Tree

For effective comparison of arrays of items. A bBST where leaf nodes store hash of its data & each non-leaf node takes and concatenates the values of children and hash, which is stored as its own value. Same value at root indicates everything in subtree identical

Example



Tutorials

Tut. 03 – Order Maintenance

~ insertAfter(A, B): by insert B as successor of A in bBST, then rotate. Similar process for insertBefore(A, B)
~ isAfter(A, B): to check if B after A in tree, compare their ranks

Tut. 04 – kd-Tree

A tree structure that is used to describe points in a 2D plane. Each node is a (x, y) pair denoting a point and it divides horizontally or vertically (alternate by levels in tree) points into left and right children.
~ search: O(logn) - recursive alternate binary search on x- / y-values.
~ construct tree from unsorted array: O(nlogn) – QuickSelect median, partition.
~ smallest x / y:

$$T(n) = 2T(n/4) + O(1)$$
$$= O(\sqrt{n})$$

Tut. 04 – Finger Search

Augmented bBST that supports search from one node to another (nearby one)
~ Use a B-Tree where each node has a parent point and is linked to nodes of same level. Record min & max of all values in subtree at each node.
~ search(x,y): from x to y (assume y>x), check whether to go up or go right neighbor (go right only when y in the neighbor's min-max range). Once go right or arrive at root, search for y normally in the subtree
  ~ Time: O(logd), d: diff. in x and y's rank.

Miscellaneous