CS2040 Notes

Koh Jia Xian

April 9, 2021

Contents

1 Definitions

1.1 Time and Space Complexity

- Space Complexity = **Total** space ever allocated
- Amortized cost $T(n)$ if $\forall k \in \mathbb{Z}$, cost of operation is $\leqq kT(n)$

1.1.1 Big O

$T(n) = O(f(n))$ if:

1. There exists a constant $c > 0$
2. and a constant $n_0 > 0$

such that for all $n > n_0$,

$$\boldsymbol{T(n) \leq cf(n)}$$

ie) An upper bound above a certain size n; Always try to get the tightest bound

1.1.2 Big Omega

$T(n) = \Omega(f(n))$ if:

1. There exists a constant $c > 0$
2. and a constant $n_0 > 0$

such that for all $n > n_0$,

$$\boldsymbol{T(n) \geq cf(n)}$$

ie) A lower bound above a certain size n

1.2 Pre and Post-conditions

Precondition	Fact that is true when the function begins
Postcondition	Fact that is true when the function ends

1.3 Invariants

Invariants	Relationship between variables that is **always true**.
Loop Invariants	Relationship between variables that is true at the beginning (or end) of each iteration of a loop.

1.4 Stability and In-Place sorting

When 2 of the same keys are sorted:

- If its value becomes out of order, **Unstable**
- **Stability: Preserving order of repeated elements**

  General Rule-of-Thumb, if got swap here-swap there (ie **NOT IN-PLACE**), it is unstable

1.5 Probability and Expected Value

- $E[X] = e_1p_1 + e_2p_2 + ... + e_kp_k$
- $E(A + B) = E(A) + E(B)$

1.6 Trees

Successor Next largest value in the tree.

Height Number of edges on longest path from root to leaf.

- $h(v) = 0$ if v is a leaf
- $h(v) = max(h(v.left), h(v.right)) + 1$

2 Common Time Complexities

Recurrence	Complexity	Remarks
$T(n) = 2T(n/2) + O(n)$	$O(nlogn)$	Height of logn, n each 'level'
$T(n) = T(n/2) + O(1)$	$O(logn)$	Height of logn, 1 each 'level'
$T(n) = 2T(n/2) + O(1)$	$O(n)$	1, 2, 4, ... n: Sum of GP
$T(n) = T(n/2) + O(n)$	$O(n)$	n, n/2, n/4 ... 1: Sum of GP

2.1 AP GP Sums

For AP, $S_n = \frac{1}{2}n(a_1 + a_n)$

- If AP is 1, 2,...n, $S_n = \frac{n^2+n}{2} = O(n^2)$

For GP, $S_n = \frac{a(r^n-1)}{r-1} = \frac{a(1-r^n)}{1-r}$

- Sum to ∞ $S_\infty = \frac{a}{1-r}$
- If GP is 1, 2, 4...n, where a = n, $r = 1/2$, $S_n = \frac{a}{1-r} = \frac{n}{1-0.5} = O(n)$

3 Binary Search

For a **sorted** array, take middle, compare to key: search LHS or RHS of mid.

```
int search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end-begin)/2;
        if key <= A[mid] then
            end = mid
        else begin = mid+1
    return (A[begin]==key) ? begin : -1
```

Functionality	• If element not in array, return index • If element not in array, return -1
Precondition	• Array is of size n • Array is sorted
Postcondition	If element is in the array: A[begin] = key
Invariant (Correctness)	$A[begin] \leq key \leq A[end]$ • *The key is in the range of the Array*
Invariant (Speed)	$(end - begin) \leq n/2^k$ in iteration k

Not just for searching Arrays:

1. • Assuming a complicated function,
   • Assume function is always increasing: $complicatedFunction(i) < complicatedFunction(i+1)$
   • $\therefore$ Find minimum value j such that $complicatedFunction(j) > 100$
2. Peak Finding (1 or 2 Dimensions)
3. QuickSelect

3.1 Peak Finding

Want to find an index i such that $\boldsymbol{arr[i] \geq arr[i-1]}$ & $\boldsymbol{arr[i] \leq arr[i+1]}$

```
FindPeak(A, n)
    //Recurse on right
    if A[n/2+1] > A[n/2] then
        FindPeak(A[n/2+1..n], n/2)

    //Recurse on left
    else if A[n/2{1] > A[n/2] then
        FindPeak(A[1..n/2-1], n/2)

    else A[n/2] is a peak; return n/2
```

Functionality	On an unsorted array, find A peak: local minimum or maximum (not a specific key)
Invariants (Correctness)	• There exists a peak in the range $[begin, end]$ Every peak in $[begin, end]$ is a peak in $[1, n]$.
Running Time	$T(n) = T(n/2) + \theta(1)$ Recurse for $log2(n)$ times $\therefore O(logn)$

3.2 Steep Peaks

Want to find a peak such that its left and right side are **strictly lower than it**.

Functionality	On an unsorted array, find A peak: local minimum or maximum (not a specific key) **If both sides are the same as mid, recurse both sides**
Running Time	$T(n) = \mathbf{2}T(n/2) + \theta(1)$ $= 16T(n/16) + 8 + 4 + 2 + 1$ ... $= nT(1) + n/2 + n/4 + ... + 1$ $\boldsymbol{= O(n)}$ **Sum of Geometric Progression**

3.3 QuickSelect

Find kth smallest element

Makes use of QuickSort's partition to ensure that the kth smallest element is before or after the randomly selected pivot

```
Select(A[1..n], n, k)
    if (n == 1) then return A[1];
    else Choose random pivot index pIndex.
        p = partition(A[1..n], n, pIndex)
        if (k == p) then return A[p];
        else if (k < p) then
            return Select(A[1..p{1], k)
        else if (k > p) then
            return Select(A[p+1], k { p)
```

Recurrence: $T(n) = T(n/2) + O(n)$

Time Complexity: $\boldsymbol{O(n)}$ (Sum of G.P.)

3.3.1 Paranoid Select

Repeatedly partition until at least n/10 in each half of partition
$E[T(n)] \leq E[T(9n/10)] + E[numofpartitions](n)$
$\leq E[T(9n/10)] + 2n$
$\leq O(n)$

4 Sorting

4.1 Bubble Sort

Iteratively swap largest values to the top.

```
BubbleSort(A, n)
    repeat (until no swaps) :
        for j <- 1 to n-1
            if A[j] > A[j+1] then swap(A[j], A[j+1])
```

Loop Invariant	At the end of iteration j, the biggest j items are correctly sorted in the **final j positions** of the array.
Invariant (Correctnness)	Sorted after n iterations
Running Time	
• Best Case	$O(n)$ [Already Sorted]
• Average Case	$O(n^2)$
• Worst Case	$O(n^2)$ [n iterations]
Space Consumption	$O(1)$
Stability	**Stable**, only swap elements that are different

4.2 Selection Sort

Find minimum element and swap it directly with the front.

```
SelectionSort(A, n)
    for j <- 1 to n-1:
        find minimum element A[j] in A[j..n]
        swap(A[j], A[k])
```

Loop Invariant	At the end of iteration j: the smallest j items are correctly sorted in the **first j positions** of the array.
Running Time	$n + (n-1) + (n-2) + ... + 1$ $= \frac{n(n-1)}{2}$ (Sum of A.P.) $= O(n^2)$
• Best Case	$O(n^2)$ [If already Sorted, will swap anyway]
• Average Case	$O(n^2)$
• Worst Case	$O(n^2)$ [n swaps]
Space Consumption	$O(1)$
Stability	**Unstable**, swap changes order

4.3 Insertion Sort

Iteratively swaps the current element into its rightful place in the sorted left side of the array.

```
InsertionSort(A, n)
    for j <- 2 to n
        key <- A[j]
        i <- j-1
        while (i > 0) and (A[i] >key)
            A[i+1] <- A[i]
            i <- i-1
        A[i+1] <- key
```

Loop Invariant	At the end of iteration j: the **first j items** in the array are in sorted order.
Running Time	$1 + 2 + 3 + ... + n$ $= \frac{n(n-1)}{2}$ (Sum of A.P.) $= O(n^2)$
• Best Case	$O(n)$ [Already Sorted]
• Average Case	$O(n^2)$
• Worst Case	$O(n^2)$ [Inverse Sorted]
Space Consumption	$O(1)$
Stability	**Stable**, swap doesn't change order, as long as implemented properly (**A[i]** $>$ **key**)

Insertion Sort can be fast(er than MergeSort!) if **List is mostly sorted**

4.4 MergeSort

Divide-and-Conquer, sort two halves, merge two sorted halves

```
MergeSort(A, n)
    if (n=1) then return;                                          //O(1)
    else:
        X <- MergeSort(A[1..n/2], n/2);                            //T(n/2)
        Y <- MergeSort(A[n/2+1, n], n/2);                          //T(n/2)
        return Merge (X,Y, n/2); //2 sorted halves combined together    //O(n)
```

Running Time	
Running Time of Merge	Given A and B of sizes n/2, $\boldsymbol{O(n)}$ to move each element back into list $\therefore T(n) = O(1)$ (if $n = 1$) $= 2T(n/2) + cn$ (if $n > 1$) $\therefore$ Height of recursion tree $h = logn$, every level cn operations $\therefore T(n) = cnlogn$, $\boldsymbol{O(n) = nlogn}$
• Best Case	$O(nlogn)$
• Average Case	$O(nlogn)$
• Worst Case	$O(nlogn)$
Space Consumption	$\boldsymbol{O(n)}$ [Using 1 temporary array, Switch the order of A and B at every recursive call.]
Stability	**Stable**

MergeSort can be slower for **Smaller number of items to sort**

4.5 QuickSort

Separate larger and smaller than a chosen **pivot** (Partitioning), recursively sort both sub-arrays.

```
QuickSort(A[1..n], n)
   if (n==1) then return;
   else
       Choose pivot index pIndex   //How?
       p = partition(A[1..n], n, pIndex)
       x = QuickSort(A[1..p-1], p-1)
       y = QuickSort(A[p+1..n], n-p)

//Returns the index of the pivot
partition(A[1..n], n, pIndex)       // Assume no duplicates, n>1
    pivot = A[pIndex];              // pIndex is the index of pivot
    swap(A[1], A[pIndex]);          // store pivot in A[1]
    low = 2;                        // start after pivot in A[1]
    high = n+1;                     // Define: A[n+1] = Infinity
    while (low < high)
        while (A[low] < pivot) and (low < high) do low++;
        while (A[high] > pivot) and (low < high) do high{ { ;
        if (low < high) then swap(A[low], A[high]);
    swap(A[1], A[low{1]);
    return low{1;
```

Invariants	• For every $i \geq high$: $A[i] > pivot$ • For every $1 < j < low$: $A[j] < pivot$
Running Time	
Running Time of Partition	$O(n)$
• Best Case	$O(nlogn)$
• Average Case	$O(nlogn)$
• Worst Case	$O(n^2)$ [eg All elements duplicates]
Space Consumption	$O(1)$ *Extra Memory allows QuickSort to be stable*
Stability	**Unstable**

4.6 QuickSort Optimisations

4.6.1 Base Case?

- Unoptimized: Recurse to single-element arrays
- Switch to Insertion Sort for small arrays (Relies on fact that InsertionSort is fast for small arrays)
- Halt Recursion early, leaving small arrays unsorted. Then perform InsertionSort on entire array

4.6.2 3-Way Partitioning

Deal with duplicates in arrays

Option 1 **2-pass Partitioning**

1. Regular Partition
2. Pack Duplicates (of pivot) together

Option 2 **1-pass Partitioning**

- Standard Solution
- Mantain Four Regions of Array (See Fig 1)

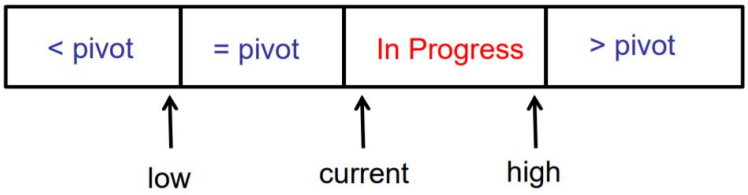

Figure 1: 1-pass Partitioning

If $bmA[current] < pivot$	low++
	Swap $A[current]$, $A[low]$
	current++
If $bmA[current] == pivot$	current++
If $bmA[current] > pivot$	Swap $A[current]$, $A[high]$
	high–

4.6.3 Choice of Pivot

In the worst case(s),

First Element	A[1]
Last Element	A[n]
Middle Element	A[n/2]
Median of first, last and middle	Median of the above 3

are equally bad, if **n executions of partition, sorting 1 element each:**

$T(n) = T(n-1) + T(1) + n$
(From Quicksort of n-1 elements + QuickSort on 1 element + Cost of partition on n elements)
$\therefore$ $\boldsymbol{O(n^2)}$ **time**.

If can choose Median: Good Performance $O(nlogn)$

If could split array (1:10) : (9:10): Good Performance $O(nlogn)$

$\therefore$ A pivot is **good** if divides array into 2 pieces, each of which is size **at least $n/10$**
Choose pivot at random: PARANOID QUICKSORT
Repeat partition until $p > (1/10)n$ and $p < (9/10)n$,
Expected number of times to choose a good pivot: $10/8 \approx 2$
$T(n) = T(n-1) + T(1) + 2n$*(Expected no. of iterations to repeat is 2)*
Hence, worst-case expected time $= \boldsymbol{O(nlogn)}$

5 Sorting Summary

Name	Best Case	Average Case	Worst Case	Extra Memory	Stable
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	Yes
Quick Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(1)$	No

5.1 Remarks

- BubbleSort vs InsertionSort: InsertionSort faster for almost-sorted arrays
- Paranoid Quicksort Worstcase: $O(nlogn)$
- **Any others?**

5.2 Invariants

Name	Invariant
Bubble Sort	At the end of iteration j, the biggest j items are correctly sorted in the **final j positions** of the array.
SelectionSort	At the end of iteration j: the smallest j items are correctly sorted in the **first j positions** of the array.
Insertion Sort	At the end of iteration j: the **first j items** in the array are in sorted order.
Merge Sort	idk lmfao probably something about at the end of iteration j of merge every 2^j group of items are in sorted order, where $2^j < n$ (???) just pulling something out of my ass :)
Quick Sort	• For every $i \geq high$: $A[i] > pivot$ • For every $1 < j < low$: $A[j] < pivot$

Recurrence	Complexity	Remarks
$T(n) = 2T(n/2) + O(n)$	$O(nlogn)$	Height of logn, n each 'level'
$T(n) = T(n/2) + O(1)$	$O(logn)$	Height of logn, 1 each 'level'
$T(n) = 2T(n/2) + O(1)$	$O(n)$	1, 2, 4, ... n: Sum of GP
$T(n) = T(n/2) + O(n)$	$O(n)$	n, n/2, n/4 ... 1: Sum of GP

5.3 AP GP Sums

For AP, $S_n = \frac{1}{2}n(a_1 + a_n)$
- If AP is 1, 2,...n, $S_n = \frac{n^2+n}{2} = O(n^2)$

For GP, $S_n = \frac{a(r^n-1)}{r-1} = \frac{a(1-r^n)}{1-r}$
- Sum to ∞ $S_\infty = \frac{a}{1-r}$
- If GP is 1, 2, 4...n, where a = n, $r = 1/2$, $S_n = \frac{a}{1-r} = \frac{n}{1-0.5} = O(n)$

6 Trees

Data Structure: Implementing a Dictionary, for eg

6.1 Binary (Search) Trees

- Binary Tree is either: 1) Empty, 2) A node pointing to 2 binary trees.
- Binary Search Trees: **All in left sub-tree** < key < **All in right sub-tree**
- **Binary Tree** is height balanced if every node in the tree is height-balanced.
- A height-balanced tree with n nodes has height $h < 2log(n)$, $\therefore O(logn)$.

Time Complexity of search(key) in BST: Height of tree
- $\boldsymbol{O(logn)}$ if balanced
- Else, worst-case $\boldsymbol{O(n)}$

6.2 Tree Traversal

In-Order: Visit left sub-tree, then SELF, then right sub-tree
Pre-Order: Visit SELF, then left sub-tree, then right sub-tree
Post-Order: Visit left sub-tree, then right sub-tree, then SELF
Level-Order Visit EVERY node at that height, then go lower level
$\boldsymbol{O(n)}$ **Time Complexity** ($\because$ Visit each node once)

6.3 Successor Finding

Basic Strategy: successor(key)
1. Search for key
2. If ($result > key$), then return result.
3. If ($result \leq key$), then search for successor of result.

```
//Search for the successor of the current TreeNode
public TreeNode successor(){
    if (rightTree != null) return rightTree.searchMin();

    TreeNode parent = parentTree;
    TreeNode child = this;
    while ((parent != null) && (child == parent.rightTree))
        child = parent;
        parent = child.parentTree;
    }

    return parent;
}
```

- $\boldsymbol{O(height)}$ **Time Complexity**

6.4 Insertion/Deletion

Insertion trivial:

If less than node, node.left == null, insert at left else recurse left.

If more than node, node.right == null, insert at right, else recurse right.

3 Cases for delete(v):	
No Children	Remove v
1 Child	Remove v, connect child(v) to parent(v)
2 Children	1. x = successor(v) 2. delete(x) (which may cause more calls of delete) 3. remove(v) 4. connect x to left(v), right(v), parent(v)

- **NOTE: Successor of deleted node has at most 1 child!** (A right node)
- ***O*(*height*) Time Complexity** (BOTH insertion and deletion)

6.5 Balance

A BST is balanced if $h = O(logn)$

How to get a Balanced Tree:

1. Define good property of tree	[AUGMENT]
2. Show that if property holds, tree is balanced.	[DEFINE BALANCE CONDITION]
3. Every insertion/deletion, make sure good property still holds:	**[INVARIANT]**
-If not, fix it	[MAINTAIN BALANCE]

6.6 AVL Trees

- Every node, store height $h = max(left.height, right.height) + 1$
- On insert & delete, update height
- node v is height-balanced if $|\boldsymbol{v.left.height - v.right.height}| \leq \mathbf{1}$ • Maintains balance using Tree-Rotations • Ma

6.6.1 Rotations

- A is LEFT-heavy if left.height > right.height
- A is RIGHT-heavy if right.height > left.height.

Assuming node v is Left-Heavy	
• v.left is balanced:	right-rotate(v)
• v.left is left-heavy:	right-rotate(v)
• v.left is right-heavy:	1. left-rotate(v.left) 2. right-rotate(v)
If v is **Right-Heavy:**	**Symmetric 3 cases**

Size of tree doesn't matter, $\therefore O(1)$ time.

6.6.2 Insertion

1. Insert tree in BST
2. Walk up tree:

- At every step, check for balance:
- If out-of-balance, use rotations to rebalance

Only need **2 Rotations** (Since in all cases, only need to reduce height of sub-tree by 1)

6.6.3 Deletion

0a. If v has no child, just delete
0b. If v has 1 child, connect child to parent
1. If v has 2 children, swap it with its successor.
2. Delete node v from binary tree (and reconnect children)

- Since successor has at most 1 (right) child, will only have to reconnect 1 node

3. For every ancestor of the deleted node:

- Check if it is height-balanced
- If not, perform a rotation
- Continue to the root

(**Deletion may take up to O(logn) rotations**)

6.6.4 Graphical Interpretation

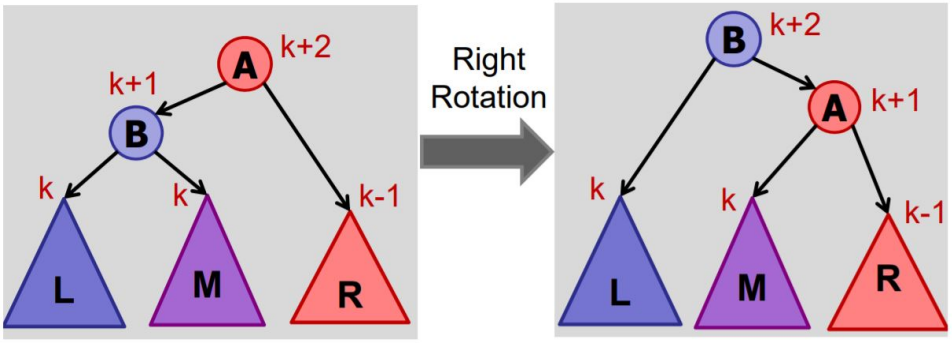

Figure 2: v.left balanced: right-rotate(v)

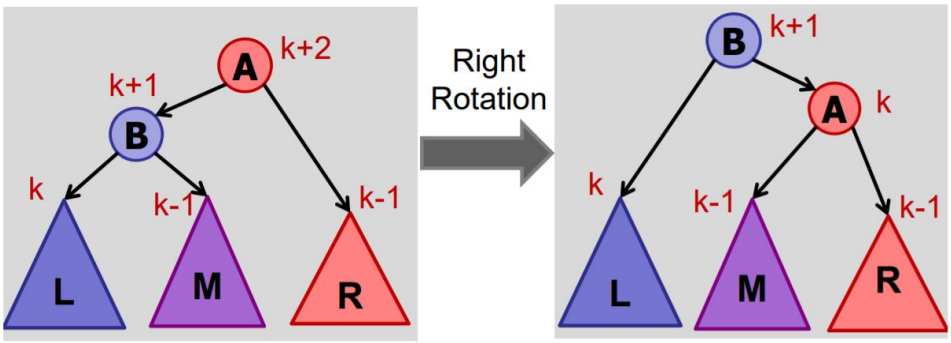

Figure 3: v.left left-heavy: right-rotate(v)

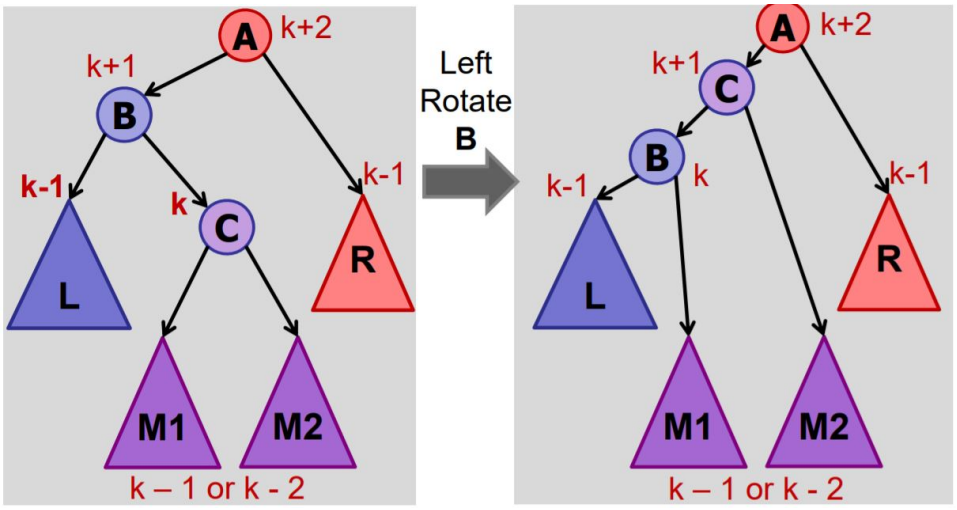

Figure 4: v.left right-heavy: First left-rotate(v.left)

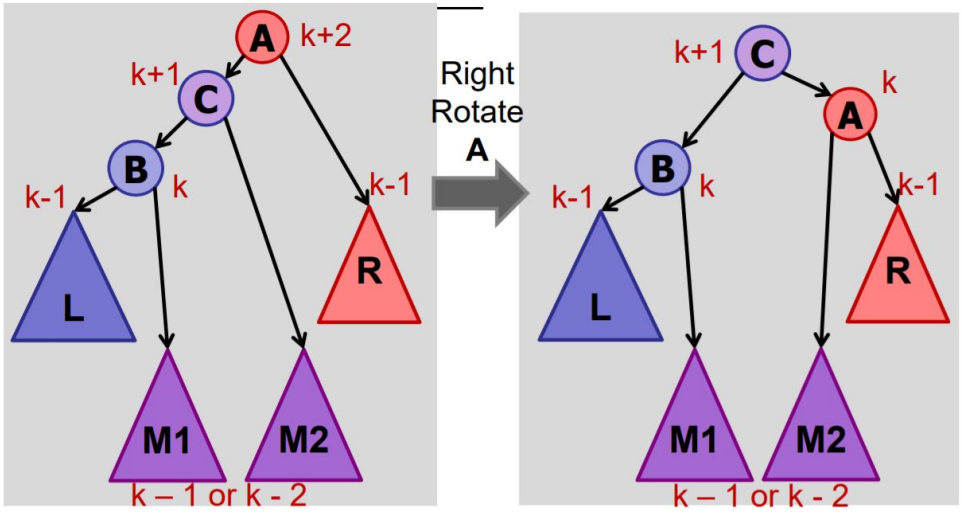

Figure 5: v.left right-heavy: then right-rotate(v)

7 Other (Augmented) Trees

7.1 Tries

Store each letter of a String as a node, using a special flag to represent the end of a word.

Cost to search a string of length L: $O(L)$

Trie tends to be faster compared to normal BST with strings

- Does not depend on size of total text
- Does not depend on number of strings (Esp if string not in trie)

Trie uses more space (in terms of more nodes)

7.2 Order Statistics

- To know the order of the node (ie rank of the key in the data structure)
- Store **size of sub-tree in every node**
- select(k): finds node with rank k
- rank(v): Computes rank at node v
- During insertion, maintain weight during rotation

```
select(k)
    rank = left.weight + 1;
    if (k == rank) then
        return v;
    else if (k < rank) then
        return left.select(k);
    else if (k > rank) then
        return right.select(k minus rank);


rank(node)
    rank = node.left.weight + 1;
    while (node != null) do
        if node is left child then
            do nothing
        else if node is right child then
            rank += node.parent.left.weight + 1;
        node = node.parent;
    return rank;
```

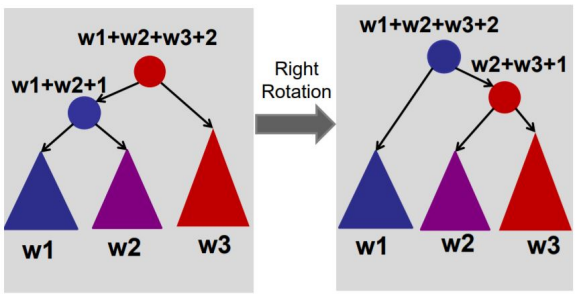

Figure 6: Update weights during insertion

7.3 Interval Trees

Find an interval containing a value

- Each node is an interval, sorted by **left endpoint**
- Each node contains the **maximum endpoint in subtree**
- Running time of search simply $O(logn)$

```
//Find interval containing x
interval-search(x)
    c = root;
    while (c != null and x is not in c.interval) do
        if (c.left == null) then
            c = c.right;
        else if (x > c.left.max) then
            c = c.right;
        else c = c.left;
    return c.interval;
```

Search find an overlapping interval, if it exists.

- If search goes right: No overlap in left-subtree

$\therefore$ **key is in right subtree or it is not in tree**

- If search goes left and no overlap, then key $<$ every interval in right sub-tree.

$\therefore$ **Either finds key in left subtree or it is not in the tree**

7.4 Range Trees/Orthogonal Range Searching

Find everyone between a certain range

- Stores all points in the **leaves** (Internal nodes store copies)
- Internal node v stores **max(v.left)**
- First find the 'split node': Is node between specified range?

∴ Do both Left and Right traversal at split node to get all nodes within range

```
FindSplit(low, high)
    v = root;
    done = false;
    while !done {
        if (high <= v.key) then v=v.left;
        else if (low > v.key) then v=v.right;
        else (done = true);
    }
    return v;


RightTraversal(v, low, high)
    if (v.key <= high) {                //Still within range
        all-leaf-traversal(v.left);
        RightTraversal(v.right, low, high);
    } else {                           //Left max larger than range, just go left
        RightTraversal(v.left, low, high);
    }


LeftTraversal(v, low, high)
    if (low <= v.key) {                 //Still within range
        all-leaf-traversal(v.right);
        LeftTraversal(v.left, low, high);
    } else {                           //Left max smaller than range, just go right
        LeftTraversal(v.right, low, high);
    }
```

- Finding split node: $O(logn)$
- Traversals recurse at most $O(logn)$ times,

outputting all (all-leaf-traversal()) is $\boldsymbol{O(k)}$, where k is number of items found.

- ∴ **Query time complexity** $\boldsymbol{= O(logn + k)}$
- Preprocessing (buildtree) time complexity: $O(nlogn)$

(Split into left and right, take highest value of left and put as key
If numofelements==1, then set as leaf)

- Space Complexity: $O(n)$
- *If just want to know the count: keep count of num of nodes in each sub-tree, and retreive that instead of all-leaf-traversal.*

Related: kd-trees (k-dimension)

8 Hashing

Standard symbol table supports:

- void insert(key, value)
- value search(key)
- void delete(key)
- bool contains(key)
- int size()

Costs of **Search** and **Insert/Delete**, and other functions required: See specifications

- AVL Tree: $O(logn)$ each
- Symbol Table: $\boldsymbol{O(1)}$ each, but extra functionality, eg Sorting ($O(nlogn)$ vs $O(n^2)$
- Symbol Table also no prede/successor queries • **Since Symbol Tables are not comparison-based**

8.1 Hash Functions & Collisions

Direct Access Tables take too much space (Number of possible keys very large)

Map keys to buckets using Hash Functions

Assume m buckets, n entries, and h is the hash function,

- 2 distinct keys **collide** if: $h(k_1) = h(k_2)$)
- Collisions **unavoidable** by Pigeonhole Principle (Table Size < Universe Size)

8.2 Collision Handling: Chaining

Put both items in same bucket, using linked List of items.

Total Space:	$O(m+n)$
Insertion:	Find hash value, add to head of linked list $\therefore O(1 + cost(h))$
Search:	Find hash value, search through linked list Worst case all values go to same bucket (emphasizing importance of good hash function) $\therefore O(n + cost(h))$

8.2.1 Simple Uniform Hashing Assumption

Assume "random" mapping:

- Every key is equally likely to map to every bucket
- Keys mapped independently
- $\therefore$ **As long as enough buckets, won't get too many keys in one bucket**

If $X(i,j) = 1$ if item i is put in bucket j, and 0 otherwise,

- $P(X(i,j) == 1) = 1/m$
- $E(X(i,j)) = 1/m$
- Thus, expected number of items per bucket $= E(\Sigma_i X(i,b))$
  $= \Sigma_i E(X(i,b))$
  $= \Sigma_i 1/m$
  $= n/m$
- $\therefore$ **load(hashtable)** = average number of items per bucket $\boldsymbol{= n/m}$

Therefore, for a Hashtable with chaining under SUHA assumption:

Search time:	$1 + n/m$ (Hash function + linked list traversal)
• Expected	$O(1)$ (Assuming $m = \Omega(n)$ buckets, eg $m = 2n$)
• Worst-case	$O(n)$
Worst-Case Insertion:	$O(1)$ if allow duplicates, preventing duplicate requires searching
Expected max linked-list length/cost	$O(logn)$ or $\Theta(logn/loglogn)$

8.3 Collision Handling: Open-Addressing

- All data directly stored in the table, one item per slot.
- On collision, **probe sequence of buckets until empty one found**
- When $m == n$, **table is full, cannot insert any more items**; cannot search efficiently
- Redefined Hash Function: $h(key, i)$, where i = number of collisions
- **Linear Probing:** Keep checking the next bucket, $h(k, 1) + (\text{i } mod \text{ m})$

```
hash-insert(key, data)
int i = 1;
while (i <= m):                          // Try every bucket
    int bucket = h(key, i);
    if (T[bucket] == null):              // Found an empty bucket
        T[bucket] = {key, data};         // Insert key/data
        return success;                  // Return
    i++;
throw new TableFullException();          // bucket full


hash-search(key)
    int i = 1;
    while (i <= m):
        int bucket = h(key, i);
        if (T[bucket] == null) return key-not-found;        // Empty bucket!
        if (T[bucket].key == key) return T[bucket].data;    // Full bucket
        i++;
    return key-not-found;                                   // Exhausted entire table.
```

delete(key): Find key to delete, **set bucket to DELETED (A tombstone value)**

- Cannot set as NULL, since search may then fail to find a key after that bucket.
- When insert(key) comes to DELETED, **overwrite deleted cell**.

8.3.1 Properties of good Hash Functions

1. $h(key, i)$ enumerates all possible buckets

- $\forall$ bucket $j, \exists$ i : $h(key, i) = j$
- The hash function is permutation of $1...m$
- If not, may return table-full when still have space left

2. Uniform Hashing Assumption

- Every key is equally likely to be mapped to every **permutation of buckets**, independent of every other key.
- Linear Probing does NOT fulfill this criteria: **Clustering** can reach $\Theta(logn)$, ruins constant time performance

In practice though, linear probing is desirable due to caching

- **Achieved through double hashing**
- Using 2 hash functions $g(k), f(k)$, $\boldsymbol{h(k, i) = [f(k) + ig(k)] \ mod \ m}$ for some large m

Specifically, if g(k) is relatively prime to m, then h(k, i) hits all buckets

8.3.2 Performance of Open Addressing

Expected Cost = First Probe + P(collision on first probe) * Expected Cost of remaining probes
- $= 1 + (n/m)(...)$
- $= 1 + (n/m)(1 + [n - 1/m - 1][...])$
- $\leq 1 + \alpha(1 + \alpha(...))$
- $\leq 1 + \alpha + \alpha^2 + \alpha^3 + ...$
- $\leq \frac{1}{1-\alpha}$

Advantages
- Saves space
- Rarely Allocate Memory
- Better Cache performance

Disadvantages
- More sensitive to choice of hash functions
- More sensitive to load (as $\alpha \to 1$)

8.4 Resizing

Assume
- Hashing with Chaining
- SUHA

Expected Search Time: $O(1 + n/m)$
Optimal Size: $m = O(n)$

If m too big ($> 10n$), too much wasted space; if m too small ($< 2n$), too many collisions

To expand hashtable:, let $m_1 and m_2$ be old and new hashtable size
- Scan old hash table: $O(m_1)$, Initialise new table: $O(m_2)$
- Insert each element in new hashtable: $O(1) * n$
- **Total: $O(m_1 + m_2 + n)$**
- If double table size, $(n == m), m = 2m$: $O(n)$ time

To shrink hashtable:, let $m_1 and m_2$ be old and new hashtable size
- Cannot be same ratio as insert, cos there will be a point where deleting/inserting 1 shrinks/expands the table

If insert doubles the table, then for delete:
- If $(n < m/4), m = m/2$

Costs of operations:
- Inserting k elements costs $O(k)$
- $\therefore$ Insert operation: **Amortized $O(1)$**
- Search operation: **Expected $O(1)$**

9 Sets

insert(Key k), contains(Key k), delete(Key k), intersect(Set¡Key¿ s), union(Set¡Key¿ s)

9.1 Implementation using Hashtable

Takes more space to keep the entire key (to resolve collisions) in the table.

9.2 Fingerprint Hashtable

Stores bits (0 and 1) instead of the key, 0 if not present, 1 if present. No key stored in the table.

- **Collisions possible**
- Lookup operation: If key is **in**, will always report true (**No False Negatives**)
- Due to collisions, even in key not in set, may sometimes report true (**False Positives**)

Thus choosing what to store is important, based on objectives

9.2.1 Table Size vs P(False Positives)

On a lookup of n elements of table of size m,

- P(No false positive) = $(1 - 1/m)^n \approx (1/e)^{n/m}$
- P(False positive) = $1 - (1/e)^{n/m}$

Assuming we want P(false positive) at most p:

s • $n/m \leq log(\frac{1}{1-p})$

So we reduced space to 1 bit per slot, but need a bigger table to avoid collisions

9.3 Bloom Filter

Fingerprint Hashtable, but 2 hash function to **store 1 in 2 different slots**.

- Lookup: Check if both slots are 1
- Still,**No False Negatives** and possible **False Positives**

Requires 2 collisions to be a false positive, but each item take more space.

Assuming we want P(false positive) at most p:

- $n/m \leq \frac{1}{2} log(\frac{1}{1-p^{1/2}})$

Deleting elements? Consider a counter instead of 1 bit in each slot:

- On insert, counter++
- On delete, counter–

If counter gets too big, no space saving: Thus need to make collisions rare

Implementing Set functions:

- Insert, delete, query: $O(k)$
- Intersection, Bitwise AND 2 bloom filters: $O(m)$

tabitem Union, Bitwise OR 2 bloom filters: $O(m)$

10 Other Data Structures

10.1 (a, b)-trees and B-trees

a, b refer to min and (max + 1) no. of children in node, where $2 \leq a \leq (b+1)/2$
Non-leaf node must have one more child than its number of keys, its **key range:** • Keys in sorted order, $v_1, v_2, ...v_k$ • First child has key range $\leq v_1$ • Final child has key range $> v_k$ • All other children c_i, where $i \in [2, k]$ have key range $(v_{i-1}, v_i]$
All leaf nodes must be same depth
Insert: split node if contain b-1 keys (Node too big)
Delete: if deleting make node too small, merge siblings y,z if have total nodes $\leq b-1$, else share by merging and splitting

B-trees are (a, b)-trees such that $a = B$, $b = 2B$

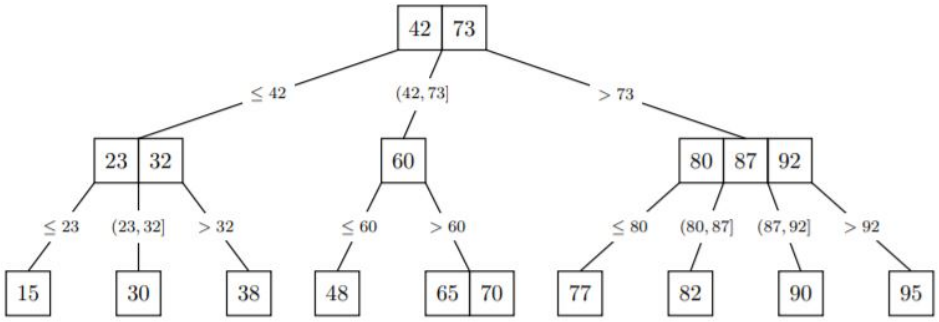

Figure 7: B-tree, where B = 2

10.2 Skip Lists

10.3 Merkle Trees

11 Graphs

Consists of at least 1 node, and unique edges that connect 2 nodes
Hypergraph: Eaah unique edge connect ≥ 2 nodes
Multigraph: Each node connected by more than 1 edge
Degree of node: Number of adjacent edges
Degree of graph: max(degree of nodes)
Diameter: Max distance between 2 nodes, following shortest path
Bipartite graph: Nodes divided to 2 sets, no edges between same set

11.1 Adjacency list

Nodes stored in an array, Edges stored as linked list per node

Memory Usage: $O(V + E)$, since of array V and size of linked lists E

Are v and w neighbours? **Fast query**
Find any neighbour of v: **Slow query**
Enumerate all neighbours: **Slow query**

11.2 Adjacency Matrix

Edges seen as pairs of nodes. For a graph with n nodes, nxn array:
At A[i][j], 1 if i and j are directly connected
A^n: Length of n paths

Memory Usage: $O(V^2)$

Are v and w neighbours? **Slow query**
Find any neighbour of v: **Fast query**
Enumerate all neighbours: **Fast query**

Generally, if graph is dense, use an adjacency matrix, if not then adjacency list

12 Graph Traversal

Start at vertex s, ends at vertex t, or visit all nodes in the graph. (Assume adjacency list)

12.1 Breadth-First Search

- **Finds shortest paths**
- Skip already visited nodes, calculate level[i] from level[i-1]

```
//Or can use a Queue to pop the earlier ones first
BFS(Node[] nodeList) {
    boolean[] visited = new boolean[nodeList.length];
    Arrays.fill(visited, false);

    int[] parent = new int[nodelist.length];
    Arrays.fill(parent, -1);

    // To make sure you visit all components
    for (int start = 0; start < nodeList.length; start++) {
        if (!visited[start]){
            Bag<Integer> frontier = new Bag<Integer>;
            frontier.add(startId);

            // Main code
            while (!frontier.isEmpty()){
                Collection<Integer> nextFrontier = new ... ;
                for (Integer v : frontier) {
                    for (Integer w : nodeList[v].nbrList) {
                        if (!visited[w]) {
                            visited[w] = true;
                            parent[w] = v;
                            nextFrontier.add(w);
                        }
                    }
                }
                frontier = nextFrontier;
            }
        }
    }
}
```

Running Time: $O(V + E)$ assuming adjacency list
- Every vertex v = start once, and added to nextFrontier once

(After visited, never re-added: $O(V)$)
- Each v.nbrList enumerated once: $O(E)$

Shortest path is a tree - Parent pointers store shortest path

12.2 Depth-first search

13 Data Structures Summary

13.1 Trees

Name	Search	Insert	Delete	Remarks
BST	$O(height)$	$O(height)$	$O(height)$	$h < 2log(n)$
AVL	$O(logn)$	$O(logn)$+ 2 rotations	$O(logn)$ + logn rotations	If v is left-heavy, - v.left is balanced/left-heavy: right-rotate(v) - v.left is right-heavy: left-rotate(v.left), right-rotate(v)
Trie	$O(length)$	$O(length)$	$O(length)$	
(a,b)-trees **B-trees**	$O(logn)$	$O(logn)$	$O(logn)$	Insert: split Delete: Merge, or Share (merge + split)

13.2 Augmented Trees

Assuming augmented from AVL, search(), insert() and delete() are $O(logn)$

Order Statistics	**Find order/rank of nodes** • Store size of sub-tree in every node • During insertion, maintain weight during rotation
Interval Tree	Nodes sorted by left endpoint Nodes contain max endpoint in tree rooted at node
Orthogonal Range Searching (kd-trees)	**Find everything within certain range** • Points stored in leaves • internal node stores max(node.left) • kd-trees: Alterate splitting between dimensions: • Query: $O(\sqrt{n}+k)$, Space: $O(n)$, Build: $O(nlogn)$
Range Tree	**Build x-tree using only x-coords, x-node contains y-tree (etc)** • Query: $O(log^2 n+k)$, Space: $O(nlogn)$, Build: $O(nlogn)$

13.3 Hashing

Assuming m is number of buckets, n is number of keys, h is cost of hash function,

Name	Search	Insert	Space	Remarks
Chaining	$O(h+n/m)$ $= O(1)$ (Expected) $= O(n)$ (Worst-case)	$O(h+1)$	$O(m+n)$	• Simple Uniform Hashing Assumption • load $= n/m$
Open-Addressing	$O(1)$	$1/(1-load)$	$O(n)$	• Uniform Hashing Assumption • Redefine hash function: Linear Probing or otherwise • Double-hashing: $h(k,i) = [f(k) + ig(k)]\ mod$ m • Tombstone value for deleted items • Performance degrades as load $= n/m$ tends to 1