

CS2040 Notes

Koh Jia Xian

February 27, 2021

Contents

1	Definitions	3
1.1	Time and Space Complexity	3
1.1.1	Big O	3
1.1.2	Big Omega	3
1.2	Pre and Post-conditions	3
1.3	Invariants	3
1.4	Stability and In-Place sorting	3
1.5	Probability and Expected Value	3
1.6	Trees	4
2	Binary Search	5
2.1	Peak Finding	6
2.2	Steep Peaks	6
2.3	QuickSelect	7
2.3.1	Paranoid Select	7
3	Sorting	8
3.1	Bubble Sort	8
3.2	Selection Sort	8
3.3	Insertion Sort	9
3.4	MergeSort	9
3.5	QuickSort	10
3.6	QuickSort Optimisations	10
3.6.1	Base Case?	10
3.6.2	3-Way Partitioning	11
3.6.3	Choice of Pivot	11
4	Sorting Summary	12
4.1	Remarks	12
4.2	Invariants	12
5	Trees	13
5.1	Binary (Search) Trees	13
5.2	Tree Traversal	13
5.3	Successor Finding	13
5.4	Insertion/Deletion	14
5.5	Balance	14
5.6	AVL Trees	15
5.6.1	Rotations	15
5.6.2	Insertion	15
5.6.3	Deletion	15
5.6.4	Graphical Interpretation	16

6	Other (Augmented) Trees	18
6.1	Tries	18
6.2	Order Statistics	18
6.3	Interval Trees	19
6.4	Range Trees/Orthogonal Range Searching	20
6.5	B-trees	21

1 Definitions

1.1 Time and Space Complexity

Space Complexity = **Total** space ever allocated

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 $n > n_0$,

$$T(n) \leq cf(n)$$

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 $n > n_0$,

$$T(n) \geq cf(n)$$

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

$$E[X] = e_1p_1 + e_2p_2 + \dots + e_kp_k$$

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 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	<ul style="list-style-type: none">• If element not in array, return index• If element not in array, return -1
Precondition	<ul style="list-style-type: none">• Array is of size n• Array is sorted
Postcondition	If element is in the array: $A[\text{begin}] = \text{key}$
Invariant (Correctness)	$A[\text{begin}] \leq \text{key} \leq A[\text{end}]$ <ul style="list-style-type: none">• <i>The key is in the range of the Array</i>
Invariant (Speed)	$(\text{end} - \text{begin}) \leq n/2^k$ in iteration k

Not just for searching Arrays:

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

2.1 Peak Finding

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

```
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] > A[n/2+1] then
        FindPeak(A[1..n/2], 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)	<ul style="list-style-type: none"> 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 $\log_2(n)$ times $\therefore O(\log n)$

2.2 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	$T(n) = 2T(n/2) + \theta(1)$ $= 16T(n/16) + 8 + 4 + 2 + 1$ \dots $= nT(1) + n/2 + n/4 + \dots + 1$ $= O(n)$ Sum of Geometric Progression

2.3 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..n], k - p)
```

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

Time Complexity: $O(n)$ (Sum of G.P.)

2.3.1 Paranoid Select

Repeatedly partition until at least $n/10$ in each half of partition

$$\begin{aligned} E[T(n)] &\leq E[T(9n/10)] + E[\text{numofpartitions}](n) \\ &\leq E[T(9n/10)] + 2n \\ &\leq O(n) \end{aligned}$$

3 Sorting

3.1 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 (Correctness)	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

3.2 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	$n + (n - 1) + (n - 2) + \dots + 1$ $= \frac{n(n-1)}{2}$ (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

3.3 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	$1 + 2 + 3 + \dots + n$ $= \frac{n(n-1)}{2}$ (Sum of A.P.) $= O(n^2)$
<ul style="list-style-type: none"> • Best Case • Average Case • Worst Case 	$O(n)$ [Already Sorted] $O(n^2)$ $O(n^2)$ [Inverse Sorted]
Space Consumption	$O(1)$
Stability	Stable , swap doesn't change order, as long as implemented properly ($A[i] > \text{key}$)

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

3.4 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 $n/2$, $O(n)$ 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 $h = \log n$, every level cn operations $\therefore T(n) = cn \log n$, $O(n) = n \log n$
<ul style="list-style-type: none"> • Best Case • Average Case • Worst Case 	$O(n \log n)$ $O(n \log n)$ $O(n \log n)$
Space Consumption	$O(n)$ [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**

3.5 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]);
  return low;
```

Invariants	<ul style="list-style-type: none">• 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	$O(n \log n)$
• Average Case	$O(n \log n)$
• Worst Case	$O(n^2)$ [eg All elements duplicates]
Space Consumption	$O(1)$ <i>Extra Memory allows QuickSort to be stable</i>
Stability	Unstable

3.6 QuickSort Optimisations

3.6.1 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

3.6.2 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
- Maintain Four Regions of Array (See Fig 1)

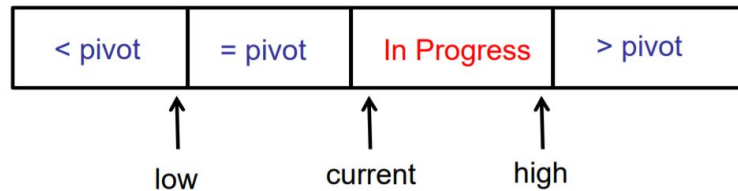


Figure 1: 1-pass Partitioning

```

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

```

3.6.3 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)
 $\therefore O(n^2)$ time.

If can choose Median: Good Performance $O(n \log n)$

If could split array (1:10) : (9:10): Good Performance $O(n \log n)$

\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 = **$O(n \log n)$**

4 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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(1)$	No

4.1 Remarks

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

4.2 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	<ul style="list-style-type: none"> • For every $i \geq high : A[i] > pivot$ • For every $1 < j < low : A[j] < pivot$

5 Trees

Data Structure: Implementing a Dictionary, for eg

5.1 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 $h < 2\log(n)$, $\therefore O(\log n)$.

Time Complexity of search(key) in BST: Height of tree

- $O(\log n)$ if balanced
- Else, worst-case $O(n)$

5.2 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

$O(n)$ **Time Complexity** (\because Visit each node once)

5.3 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;
}
```

- $O(\text{height})$ Time Complexity

5.4 Insertion/Deletion

Insertion trivial:

If less than node, $\text{node.left} == \text{null}$, insert at left else recurse left.

If more than node, $\text{node.right} == \text{null}$, 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	<ol style="list-style-type: none">1. $x = \text{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)
- $O(\text{height})$ Time Complexity (BOTH insertion and deletion)

5.5 Balance

A BST is balanced if $h = O(\log n)$

How to get a Balanced Tree:

1. Define good property of tree
2. Show that if property holds, tree is balanced.
3. Every insertion/deletion, make sure good property still holds:
-If not, fix it

[AUGMENT]

[DEFINE BALANCE CONDITION]

[INVARIANT]

[MAINTAIN BALANCE]

5.6 AVL Trees

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

5.6.1 Rotations

- A is LEFT-heavy if $\text{left.height} > \text{right.height}$
- A is RIGHT-heavy if $\text{right.height} > \text{left.height}$.

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.

5.6.2 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)

5.6.3 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 $O(\log n)$ rotations)

5.6.4 Graphical Interpretation

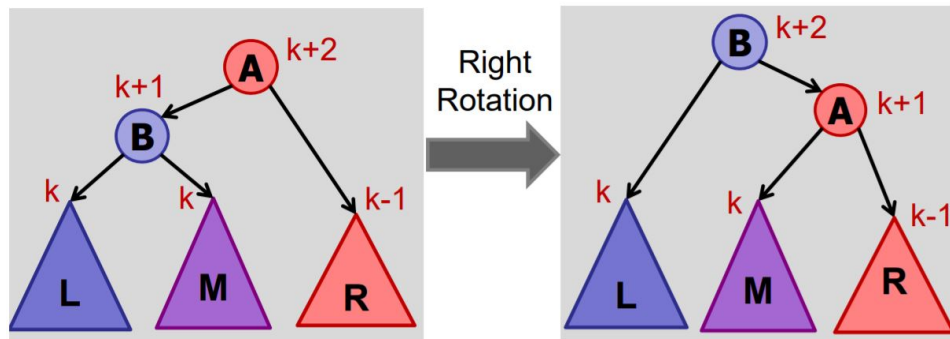


Figure 2: v.left balanced: right-rotate(v)

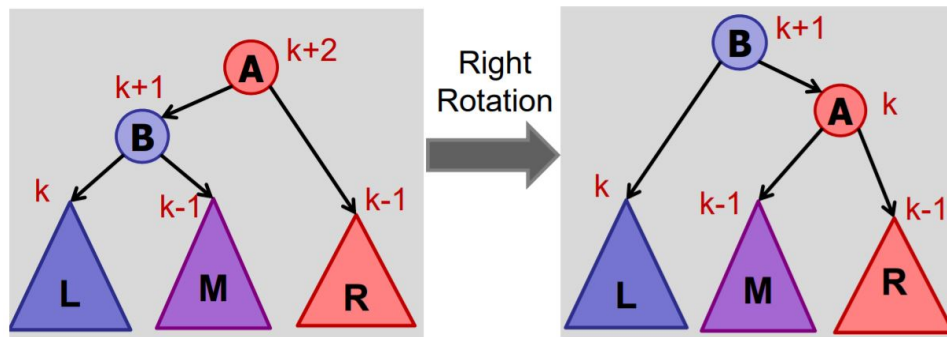


Figure 3: v.left left-heavy: right-rotate(v)

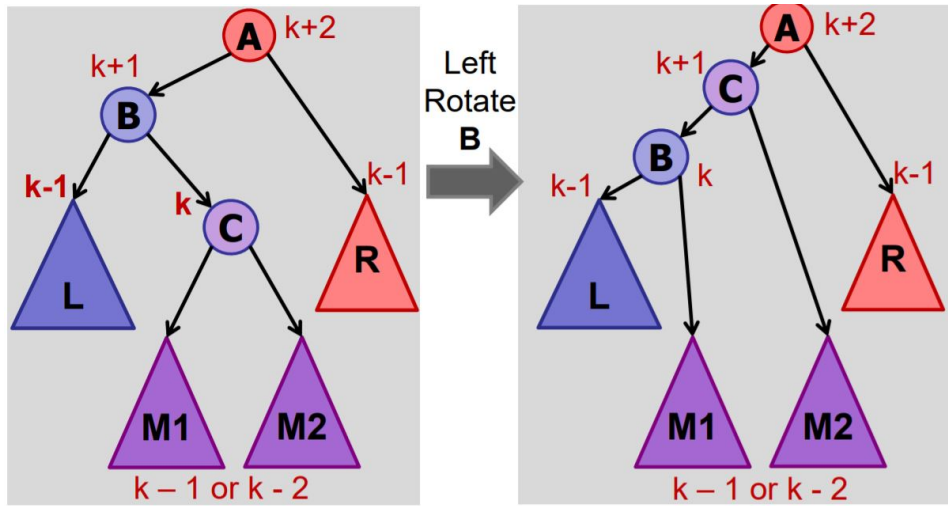


Figure 4: v.left right-heavy: First left-rotate(v.left)

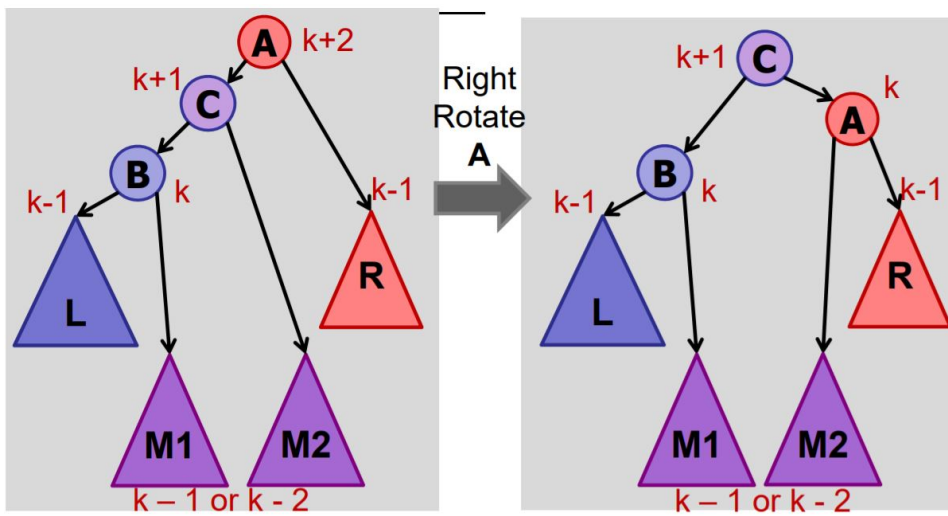


Figure 5: v.left right-heavy: then right-rotate(v)

6 Other (Augmented) Trees

6.1 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)

6.2 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-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;
```

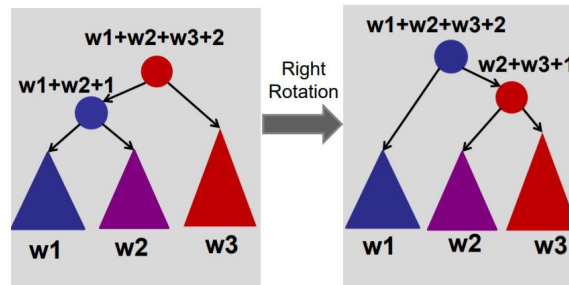


Figure 6: Update weights during insertion

6.3 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 $O(\log n)$

```
//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
∴ **key is in right subtree or it is not in tree**
- If search goes left and no overlap, then $\text{key} < \text{every interval in right sub-tree}$.
∴ **Either finds key in left subtree or it is not in the tree**

6.4 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: $O(\log n)$
- Traversals recurse at most $O(\log n)$ times, outputting all (all-leaf-traversal()) is $O(k)$, where k is number of items found.
- ∴ **Query time complexity** = $O(\log n + k)$
- Preprocessing (buildtree) time complexity: $O(n \log n)$
(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 retrieve that instead of all-leaf-traversal.*

Related: kd-trees (k-diimension)

6.5 B-trees