# CS2040S Part 1

Fingerprints Hash Table(FHT - Set ADT)
⌨️ Pseudo-Code for insert & query

# W2L1 Searching

## Asymptotic Notations

### Asymptotic Notations

| Aa Notation | Equation | Condition | Note |
|---|---|---|---|
| Big-O | $T(n) = O(f(n))$ | if there exists a constant c > 0, there exists a constant n0 > 0 such that T(n) ≤ cf(n) | Upper Bound |
| Big-Ω | $T(n) = \Omega(f(n))$ | if there exists a constant c > 0, there exists a constant n0 > 0 such that T(n) ≥ cf(n) | Lower Bound |
| Big-$\Theta$ | $T(n) = \Theta(f(n))$ | iff $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$ | Lower Bound<br>Upper Bound |

### Big-O Order

| Aa Function | Tags |
|---|---|
| $5$ | Constant |
| $log(log(n))$ | Double Log |
| $log(n)$ | Log |
| $log^2(n)$ | Polylogarithmic |
| $n$ | Linear |
| $log(n!)$ | |
| $nlog(n)$ | Log-linear |
| $n^3$ | Polynomial |
| $n^3log(n)$ | |
| $n^4$ | Polynomial |
| $2^n$ | Exponential |
| $(2^2)^n$ | |
| $n!$ | Factorial |

### Runtime Complexity Rules

| Aa Rule | Example |
|---|---|
| If $T(n) $ is a polynomial degree k, then $T(n)=O(n^k)$ | $10n^2 + 50n^3 = O(n^5)$ |
| If $T(n) =O(f(n))$ and $S(n) = O(g(n))$, then $T(n) + S(n) = O(f(n)+g(n))$ | $10n^2 + 5nlog(n) = O(n^2 + nlog(n)) = O(n^2)$ |

| Aa Rule | Example |
| --- | --- |
| If $T(n)=O(f(n))$ and $S(n) =O(g(n))$, then $T(n)*S(n)=O(f(n)*g(n))$ | $(10n^2)(5n) = 50n^3 = O(n * n^2) = O(n^3)$ |
| If $T(n)=O(f(n))$ and $S(n) =O(g(n))$, then $T(S(n))=O(f(g(n)))$ (similar to fxn composition) | |
| If $T(n)=O(f(n))$ and $S(n) =O(g(n))$, then $T(n)^{S(n)}\neq O(f(n)*g(n))$ | |

# Algorithm Analysis

## 🔁Loops

cost = (# iterations)(max cost of 1 iteration)

```
int sum(int k, int[] intArray) {
    int total=0;
    for (int i=0; i<= k; i++){
      total = total + intArray[i];
    }
    return total;
}
```

## 🔁Nested Loops

cost = (# iterations)(max cost of 1 iteration)

```
int sum(int k, int[] intArray) {
  int total=0;
  for (int i=0; i<= k; i++){
    for (int j=0; j<= k; j++){
      total = total + intArray[i];
    }
  }
  return total;
}
```

## 🔢Sequential Statements

cost = (cost of 1st) + (cost of 2nd)

```
int sum(int k, int[] intArray) {
  for (int i=0; i<= k; i++)
    intArray[i] = k;
  for (int j =0; j<= k; j++)
    total = total + intArray[i];
  return total;
}
```

## ⁉️If-Else Statements

cost = max(cost of 1st, cost of 2nd) ≤ (cost of 1st + cost of 2nd)

```
void sum(int k, int[] intArray) {
  if (k > 100)
    doExpensiveOperation();
  else
    doCheapOperation();
  return;
}
```

## 🤨Recurrences

e.g. Fibonacci

$$T(n) = 1 + T(n-1) + T(n-2) = O(2^n)$$

```
int fib(int n) {
  if (n <= 1)
    return n;
  else
    return fib(n-1) + fib(n-2); // T(n-1) + T(n-2)
}
```

## Examples

```
public static boolean isPrime(int n) {
    if (n<2) {
        return false;
    }
    for (int i=2;i<n;i++){
        if (isDivisible(n,i)) {
            return false;
        }
    }
    return true;
}
```

Time O(n) — Inclusive end - inclusive start + 1
Space O(1) — (n-1)-2+1 = (n - 2) iterations

exclusive end - inclusive start: (n-2) iterations

Worst case: when the arg is prime (exhaust the entire loop)

What is the order of growth for `isPrime` given the following order of growths for `isDivisible(n,i)`?

1. Time: $O(1)$, Space $O(1)$
2. Time: $O(n)$, Space $O(n)$
3. Time: $O(n)$, Space $O(1)$
4. Time: $O(i)$, Space $O(1)$

| Time | Space |
|---|---|
| O(n) | O(1) |
| O(n²) | O(n) |
| O(n³) | O(i) |
| O(n!) ✗ | O(1) |

→ space dep on memory & prog lang

i takes diff values not exactly problem size

O(n³)

T(n) = 2+3+4+...+(n-1) = (n+1)(n-2)/2 = O(n²)

isDivisible: i=2 ⇒ 2 times, i=3 ⇒ 3 times, ..., i=n-1 ⇒ n-1 times

---

```
public static boolean isPrime2(int n) {
    if (n<2) {
        return false;
    }
    for (int i=2;i<sqrt(n);i++){
```

T(n) = 2+3+...+(√n-1) = O(√n)

isDivisible

1

```
            if (isDivisible(n,i)) {
                return false;
            }
        }
        return true;
    }
```

Time $O(\sqrt{n})$
space $O(1)$

What is the order of growth for `isPrime2` given the following order of growths for `isDivisible(n,i)`?

1. Time: $O(n)$, Space $O(1)$
2. Time: $O(i)$, Space $O(1)$

| Time | space |
|---|---|
| $O(n^{3/2})$ | $O(1)$ |
| $O(i\sqrt{n})$ ✗ | $O(1)$ |

$O(\sqrt{n}^2) = O(n)$ ✓

---

```
public static boolean isPrime3(int n) {
    if (n<2) {
        return false;
    }
    for (int i=2;i<=sqrt(n);i++){
        if (isPrime3(i) && isDivisible(n,i)) {
            return false;
        }
    }
    return true;
}
```

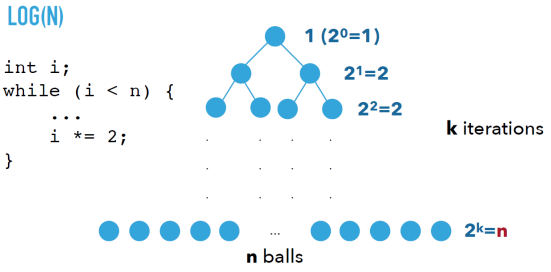
$O(1)$ if
loop: $2+3+\ldots+\sqrt{n} = O(n)$
$O(1)$ if
$O(1)$ $T(i)$ $O(1)$
$\sqrt{n}$ loops $\times (1+1)$ WD every loop
$T(n) = T(i) = T(2) + T(3) + \ldots + T(\sqrt{n}) + k\sqrt{n}$
$\leq T(n)$
↳ all the constants (if)
$T(n) \leq \sqrt{n}\,T(\sqrt{n}) + k\sqrt{n}$ expand $\Rightarrow$ $O(n)$

We are simulating the Sieve of Eratosthenes (see https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes). How does the performance of `isPrime3` compare to `isPrime2`?

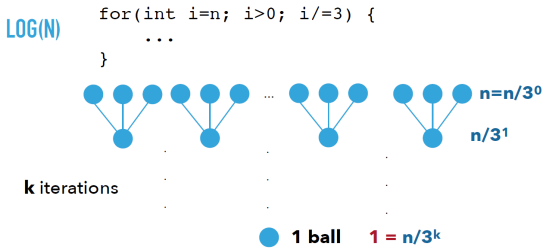
You can assume that the order of growth for `isDivisible(n,i)` is $O(1)$. $O(n)$

## Recurrence Cheat sheet

- 1 logarithmic recursive call (decreasing/increasing at constant factor) that doesn't sum to n: **O(log n)**




$\log 2^k = \log n \Rightarrow k\log 2 = \log n$




$3^k = n \Rightarrow k\log 3 = \log n$

- Multiple logarithmic recursive call that sums to n with O(1)/no work done at each level: **O(n)**
- Multiple logarithmic recursive calls that sums to n, with O(n) work done per level: **O(n log n)**
- Multiple logarithmic recursive calls that sums to $n^2$, with $O(n^2)$ work done per level: $O(n^2)$

```
int MoreQuacks(int n) {
    for (int i=0; i<n; i++) {  n
        for (int j=n+1; j<i; j++) {  0
            System.out.println("Quack.");
        }
}
```

Time: O(n) Space: O(1)

```
int WhoQuacked(int x) {
    for (int i=10000; i>=1; i/=2){  14
        System.out.println("Duck.");
    }
}
```

Time: O(1) Space: O(1)

```
int MakeMeQuack(int n) {
    if (n >= 1000) {
        System.out.println("Quack.");
        return;
    }

    int[] a = new int[n];
    MakeMeQuack(n+1);
}
```

Time: O(1) Space: O(1)

```
int RecursiveQuacks(int n) {
    if (n < 2) {
        System.out.println("Quack.");
        return 1;
    } else {
        for (j=1; j<n; j++)
            System.out.println("Quack.");
        int a = RecursiveQuacks(x/3);
        int b = RecursiveQuacks(x/3);
        int c = RecursiveQuacks(x/3);
        return(a+b+c);
    }
}
```

Time: O(nlogn) Space: O(logn)

```
int QuackWithAStack(int n, Stack s) {
    for (int i=0; i<n/2; i++) {
        s.push(i);
        s.pop(i);
        System.out.println("Quack.");
    }
}
```

n

1

Time: O(n) Space: O(1)

```
int RecursiveQuackQueue(int n, Queue q) {
    if (n < 2) {
        return n;
    }
    for (int i=0; i<2; i++) {
        q.enqueue(RecursiveQuackQueue(n/2, q));
    }
    return n;
}
```

Time: O(n)

## Binary Search

### Specification

- returns element if it is in the array
- returns "null" if it is not in the array

### Alternate Specification

- returns index if it is in the array
- returns -1 if it is not in the array

### ✅Precondition

- def: fact that is true when the function begins
- usually important for the algo to work correctly
- For Binary Search:
  - **array is of size n**
  - **array is sorted**

## ✅Postcondition

- def: fact that is true when the function ends
- usually useful to show that the computation was done correctly
- For Binary Search
  - if element is in the array: `A[begin] = key`

## ✳Invariant

- relationship between variables that is always true
- For Binary Search:
  - Correctness: `A[begin] <= key <= A[end]`
  - Performance: $(end - begin) <= n/2^k$ in iteration k

### Loop Invariant

- relationship between variables that is true at the beginning (or end) of each iteration of a loop
- For Binary Search
  - `A[begin] <= key <= A[end]` i.e. the key is in the range of the array
  - kth iteration: $(end - begin) = n/2^k \Rightarrow n/2^k = 1 \Rightarrow k = log(n)$
  - Error checking:

```
if ((A[begin] > key) || (A[end] < key)) {
  System.out.println("error");
}
```

## ⌨Pseudo-Code

```
// input sorted array: A[0...n-1]
search(A, key,n)
  begin = 0;
  end = n - 1;
  while (begin < end) do:
    int mid = begin + (end - begin) /2;
    if (key <= A[mid]) then
        end = mid;
    else
        begin = mid + 1;
  return (A[begin] == key) ? begin : -1;
// returns the index of the key in the array
```

## Tutorial Allocation Example

```
MaxStudents(x) = no. of students in most crowded tutorial if we offer x tutorials

Search(n)
```

```
  begin = 0;
  end = n - 1;
  while (begin < end) do:
    mid = begin + (end - begin)/2;
    if MaxStudents(mid) <= 15 then // 15 is the max no. of students that we should have
      end = mid;
    else
      begin = mid + 1;
  return begin;
```

# W2L2 Peak Finding

## Global Max v.s. Local Max

### ⌨️Global Max Pseudo-Code

```
Unsorted Array: A[0...n-1]

FindMax(A, n)
  max = A[1];
  for(int i = 0; i < n; i++) {
    if (A[i] > max) {
      max = A[i]
    }
  }
// Time complexity: O(n) --> too slow
```

### ⌨️Local Max Pseudo-Code

- `A[i-1] ≤ A[i]` and `A[i+1] ≤ A[i]` (note ≤ not just <)
  - e.g. [3,3,3]: all elements are considered peaks
- Assumption: `A[-1] = A[n] = -MAX_INT`

```
FindPeak(A, n)
  if A[n/2] is a peak then
    return n/2
  else if A[n/2+1] > A[n/2] then // not <=
    // Search for peak in right half.
    FindPeak(A[n/2+1..n], n/2)
  else if A[n/2–1] > A[n/2] then
    // Search for peak in left half.
    FindPeak(A[1..n/2-1], n/2)

  // else: a peak is found (not steep peak)

// Runtime: T(n) = T(n/2) + theta-1 = O(log n)
//              recursion + time for comparing A[n/2] with neighbours

// JAVA Implementation
public static int FindPeak(int[] arr, int n) {
        if (n == 1 || arr[0] > arr[1]) {
            return 0;
        } else if (arr[n - 1] > arr[n-2]) {
            return n;
        } else if (arr[n/2] > arr[n/2-1]) {
            return FindPeak(Arrays.copyOfRange(arr,n/2, n), n/2);
        } else if (arr[n/2-1] > arr[n/2]) {
            return FindPeak(Arrays.copyOfRange(arr,0,n/2),n/2);
        } else {
            return n;
        }
    }
```

### ✳Invariants

- Key property:
  - If we recurse in the right half, then there exists a peak in the right half.
- Correctness:
  - There exists a peak in the range [begin, end]
  - Every peak in [begin, end] is a peak in [1, n]

### Steep Peaks

- `A[i-1] < A[i] and A[i+1] < A[i]`
- Assumption: `A[-1] = A[n] = -MAX_INT`

```
// Following the algo to find a local max
...
if A[n/2-1] == A[n/2] == A[n/2+1] then
  Recurse on left & right sides

// Time Complexity:
// T(n) = 2T(n/2) + O(1)
//      = nT(1) + n/2 + n/4 + n/8 + … + 8 + 4 + 2 + 1
//      = O(n)
```

# W3L1 Sorting (Ascending)

## Bogo Sort

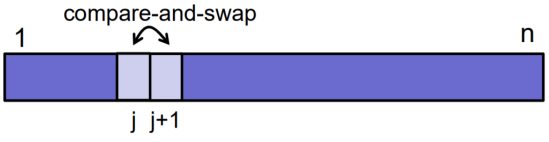
- Choose a random permutation of the array A
- If A is sorted, return A
- Time complexity: $O(n * n!)$
- Stable: No
  - random permutation may swap elements

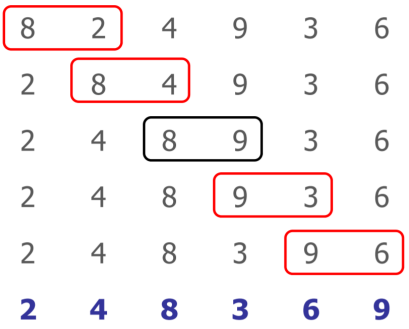
## Bubble Sort

- Compare neighbouring elements: if L > R → swap

### ⌨Pseudo-Code

```
BubbleSort(A, n)
  repeat n times: // (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.
- Correctness: after n iterations → sorted

### ⏳Time Complexity

- Worst case: $O(n^2)$
  - sorted from largest to smallest
- Best case: $O(n)$
  - sorted from smallest to largest

### 📁Space Complexity

- In-place:
  - all manipulations happens within the array
  - $O(1)$

### 💣Stability

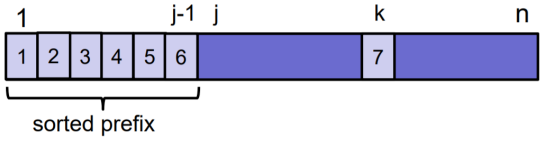
- Stable
- Only swap elements that are different

## Selection Sort

- Find the jth smallest element → swap with the jth index (for 0 ≤ j < n)

### ⌨️Pseudo-Code

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

// more detailed ver
n = A.length
for j = 1 to n-1:
  smallest = j
  for i = j+1 to n
    if A[i] < A[smallest]: smallest = i
  swap(A, j, smallest)
```



| | | | | | |
|---|---|---|---|---|---|
| 8 | **2** | 4 | 9 | 3 | 6 |
| 2 | 8 | 4 | 9 | **3** | 6 |
| 2 | 3 | **4** | 9 | 8 | 6 |
| 2 | 3 | 4 | 9 | 8 | **6** |
| 2 | 3 | 4 | 6 | **8** | 9 |
| 2 | 3 | 4 | 6 | 8 | 9 |

### ✳️Loop Invariant

- At the end of iteration j, the smallest j items are correctly sorted in the first j positions of the array.
- Initialisation: A[1,..j-1] is empty, so invariant is true
- Maintenance: Need invariant for inner loop stating that A[smallest] is the smallest element in A[j..i-1]
  - When inner loop terminates, i == n + 1, so A[smallest] is the smallest element in A[j..n]
  - if outer invariant is true before loop, it will be true after swapping on last line and incrementing j
- *Termination*: The invariant needs to imply a sorted array when the loop terminates.
  - At termination, j == n, so by the invariant, A[1..n-1] is sorted and does not have elements larger than A[n], implying that the whole array is sorted

### ⌛Time Complexity

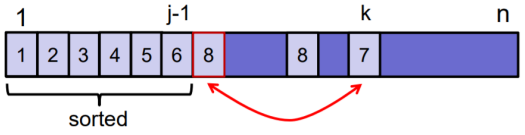
- Worst case = Best case: $O(n^2)$

### 📁Space Complexity

- In-place: $O(1)$

### 💣Stability

- NOT stable
- swap changes order



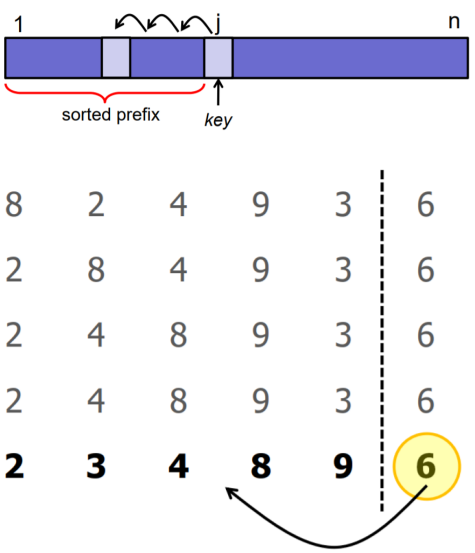

## Insertion Sort

- Starting from index 1, compare with the left (sorted) element until the current element find the index where the left element is smaller.

### ⌨️Pseudo-Code

```
InsertionSort(A, n)
  for j <- 2 to n
    key <- A[j]
    // Insert key into the sorted array A[1..j-1];
    // repeat comparison at most j times
    i <- j-1
      while (i > 0) and (A[i] > key) // stable as long as > and not >=
        A[i+1] <- A[i]
```

```
        i <- i-1
      A[i+1] <- key
```




### 🌟 Actual Code

```
Insertion(A, n)
for (int j = 1; j < n; j++) {
    int key = arr[j];
    int i = j - 1;
    while (i >= 0 && arr[i] > key) {
        arr[i+1] = arr[i];
        i --;
    }
    arr[i+1] = key;
}
```

### ✳️Loop Invariant

- At the end of iteration j, the first j items in the array are in sorted order.

### ⌛Time Complexity

- Worst case: $O(n^2)$
  - sorted from largest to smallest
- Best case: $O(n)$
  - sorted from smallest to largest

### 📁Space Complexity

- In-place: $O(1)$

### 💣Stability

- Stable as long as `>` and not `>=` is used for comparison

## Merge Sort

**Divide-and-Conquer Sorting**

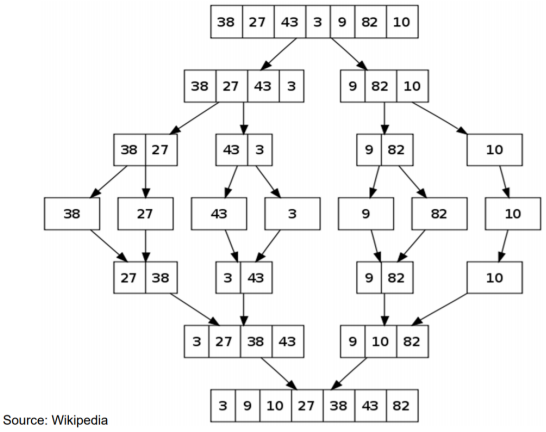
1. **Divide:** split array into 2 halves
2. **Recurse:** sort the 2 halves
3. **Combine:** merge the 2 sorted halves

**Merge**

- In each iteration, move one element to final list.
- After n iterations, all the items are in the final list.
- Each iteration takes O(1) time to compare two elements and copy one.
- Runtime: $O(n) = cn$

## ⌨️Pseudo-Code

```
/** Version 1 **/
MergeSort(A, n)
  if (n = 1) then return; // base case; O(1)
  else:
    X <- MergeSort(A[1..n/2], n/2); // recurse; T(n/2); S(n/2)
    Y <- MergeSort(A[n/2+1, n], n/2); // recurse; T(n/2); S(n/2)
  return Merge (X,Y, n/2); // combine solutions; T = O(n); S = O(n log n) (creates a temp arr of size n)

/** Version 2 **/
MergeSort(A, begin, end, tempArray)
  if (begin = end) then return;
  else:
    mid = begin + (end-begin)/2
    MergeSort(A, begin, mid, tempArray);
    MergeSort(A, mid+1, end, tempArray);
Merge(A[begin..mid], A[mid+1, end], tempArray);
Copy(tempArray, A, begin, end); // Merge copies items into tempArray, we then copy the items back into arr A
// on termination, items in range [begin, end] are sorted in A
// tempArray is used for workspace
// no new arrays are allocated during the sort
// S(n) = 2S(n/2) + O(1) = O(n)

/** Version 3 **/
// minimise copying items --> switch temporary array at every step
MergeSort(A, B, begin, end) // both A & B have copies of unsorted arr initially
  if (begin = end) then return;
  else:
    mid = begin + (end-begin)/2
    MergeSort(B, A, begin, mid);
    MergeSort(B, A, mid+1, end);
Merge(A, B, begin, mid, end); // switch the order of A & B at every recursive call
```



Source: Wikipedia

## 🌟 Actual Code

```
import java.util.Arrays;

public class Test {
    public static void merge_sort(int[] A) {
        merge_sort_helper(A, 0, A.length -1);
    }

    public static void merge_sort_helper(int[] A, int low, int high) {
        if (low < high) {
        int mid = (low + high) / 2;
            merge_sort_helper(A, low, mid);
            merge_sort_helper(A, mid + 1, high);
            merge(A, low, mid, high);
        } else{ }
    }

    public static void merge(int[] A, int low, int mid, int high) {
        int[] B = new int[A.length];
        int left = low;
        int right = mid + 1;
        int Bidx = 0;

        while (left <= mid && right <= high) {
            if (A[left] <= A[right]) {
                B[Bidx] = A[left];
                left = left + 1;
            } else{
                B[Bidx] = A[right];
                right = right + 1;
            }
            Bidx = Bidx + 1;
        }

        while (left <= mid) {
            B[Bidx] = A[left];
            Bidx= Bidx+ 1;
            left = left + 1;
        }
        // right half is exhausted
        // no more elements in the right half
        // put the remaining elements in the left list into B directly

        while (right <= high) {
            B[Bidx] = A[right];
            Bidx = Bidx + 1;
            right = right + 1;
        }
        // left half is exhausted
        // only one of the two while loops will be evaluated
        // cannot have both halves being exhausted simultaneously

        for (int k = 0; k < high - low + 1; k = k + 1) {
            A[low + k] = B[k];
        }
        // [low + k] may not be [0]
        // depends on where the half is split & merged
    }

    public static void main(String[] args) {
        int[] arr = {1,3,6,7,2,4,5};
        merge_sort(arr);
        System.out.println(Arrays.toString(arr));
    }
}
```

## ⌛Time Complexity

- needs to divide before merge even for sorted list

$T(n) = \Theta(1)$ if (n = 1)

$T(n) = 2 \cdot T(n/2) + cn$ if (n > 1)

## 📁Space Complexity

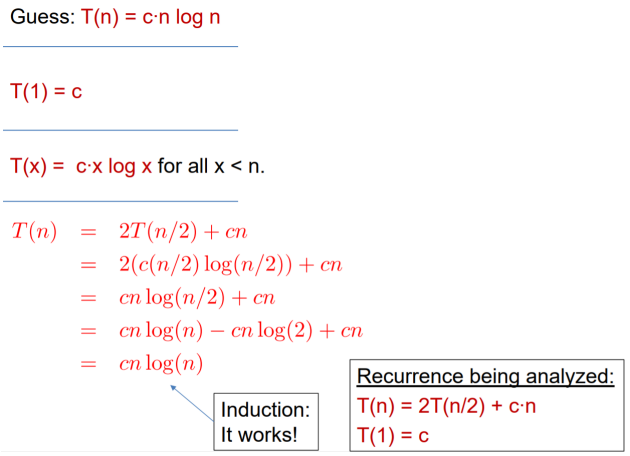
- $O(n)$
- not in-place: creates a new array with the sorted elements
- depends on `merge`
  - creates a temp array (size n) to store the sorted elements from each half
  - Worst case: $S(n) = \Theta(1)$ if (n = 1) $= 2S(n/2) + n$ if (n > 0) $= O(n \cdot log(n))$

### 💣Stability

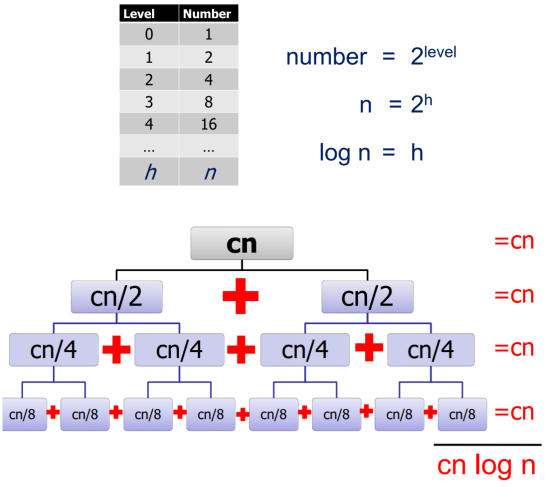
- Stable if `merge` is stable

### 💫Techniques for Solving Recurrences

1. **Guess and verify via induction**




2. **Draw the recursion tree**




3. **Master Theorem (CS3230) or the Akra-Bazzi Method, or other advanced techniques**

## Quick Sort

1. **Divide:**

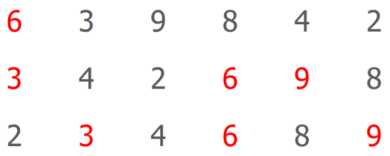
- Partition the array into 2 sub-arrays around a *pivot* `x` such that (elements in lower sub-array ≤ `x` ≤ elements in upper sub-array)


2. **Conquer:**
    - Recursively sort the two sub-arrays
3. **Combine:** trivial, do nothing

- Key: efficient `partition` sub-routine

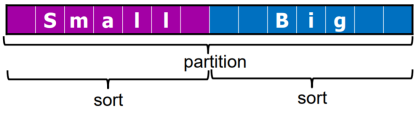
**Partition**

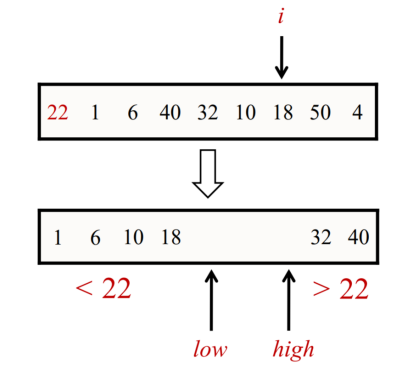
1. Choose a pivot
2. Find all elements smaller than the pivot
3. Find all elements larger than the pivot



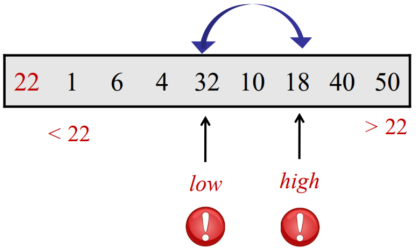

## ⌨️Pseudo-Code (distinct elements)

```
/** QuickSort Version 1 - choosing the last element as the pivot**/
QuickSort(A[1...n], n)
  if (n==1) then return;
  else
    // Choose pivot pIndex for Partition Version 2
    p = partition(A[1..n], n) // pivot index at the end
    x = QuickSort(A[1..p-1], p-1)
    y = QuickSort(A[p+1..n], n-p)


/** Version 1 new array created **/
partition(A[2..n], n, pivot)      // Assume no duplicates; T(n) = O(n)
  B = new n element array
  low = 1;
  high = n;
  for (i = 2; i<= n; i++)
      if (A[i] < pivot) then      // if curr element is smaller than pivot, start placing from the left (start)
          B[low] = A[i];
          low++;
      else if (A[i] > pivot) then // if curr element is greater than pivot, start placing from the right (end)
          B[high] = A[i];
          high-- ;
  B[low] = pivot;
  return < B, low >               // returns arr B with low as the pivot


/** Version 2 in-place **/
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++; // stops when A[low] >= pivot
        while (A[high] > pivot) and (low < high) do high--; // stops when A[high] <= pivot
        if (low < high) then swap(A[low], A[high]); // swaps the 2 elements which are not in correct pos
swap(A[1], A[low-1]);         // low == high at the end of the loop; swaps pivot and the (high-1) elem since A[high] > pivot
return low-1;                 // returns the index of the final pos of pivot
```

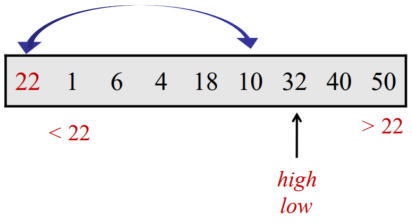



Partition Version 1




Partition Version 2 P1




Partition Version 2 P2

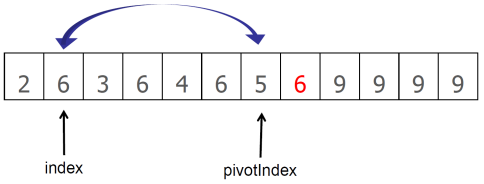
## ⌨️Pseudo-Code (duplicate elements)

```
/** QuickSort Version 2 - choosing a specific pivot index **/
// to minimise run time due to duplicates
// choosing the 1st/last/middle element could result in O(n^2) worst case
QuickSort(A[1..n], n)
  if (n==1) then return;
  else
    // Choose pivot index pIndex.
    // p = partition(A[1..n], n, pIndex)
    pIndex = random(1, n)
    p = 3wayPartition(A[1...n], n, pIndex)
    x = QuickSort(A[1..p-1], p-1)
    y = QuickSort(A[p+1..n], n-p)

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- - ;
```

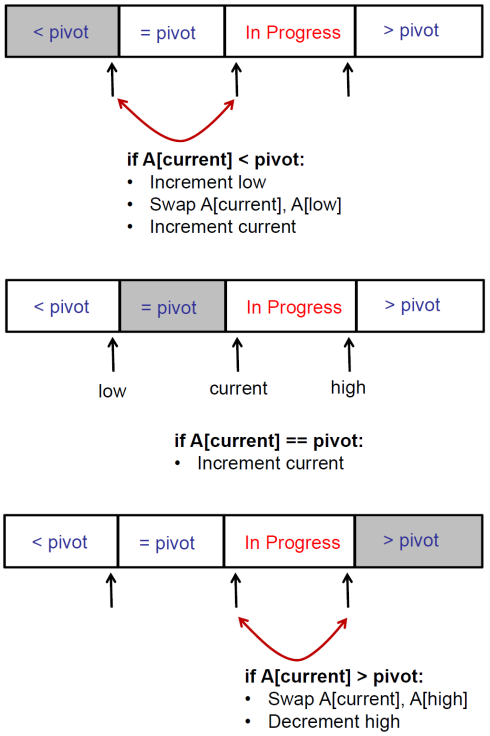
```
    if (low < high) then swap(A[low], A[high]);
  swap(A[1], A[low-1]);
  return low-1;
```

### 3-Way Partition for arrays with duplicate elements

1. Two pass partitioning
   - regular partition → pack duplicates
   - choose a `pIndex`
   - `for` loop
   - if `A[index] != A[pIndex]` then `index++`
   - if `A[index] == A[pIndex]` then `pIndex--` and `swap(A, index, pIndex)`
   - if `index == pIndex` then return




2. **One pass partitioning (standard)**
   - maintain 4 regions of the array




## ✳Partition Invariants

### Version 1 (new array) - distinct elements

- For every *i*< *low*: `B[j] < pivot`
- For every *j* > *high*: `B[j] < pivot` ⇒ `B[i] = pivot`

### Version 2 (in-place) - distinct elements

- At the end of every loop iteration,
  - `A[high] > pivot`
  - for all *i* ≥ *high*, `A[i] > pivot`
  - for all *1* < *j* < *low*, `A[j] < pivot`
- (pivot is in the correct position as the sorted array)

**One-Pass Partitioning - duplicates**

- Each region has proper elements (`< pivot`, `= pivot`, `> pivot`).
- Each iteration, In Progress region decreases by one.

## ⌛Time Complexity

- `Partition`: $O(n)$
- `QuickSort`: worst case ⇒ $O(n^2)$; best case ⇒ $O(n \cdot log(n))$

## 📁Space Complexity

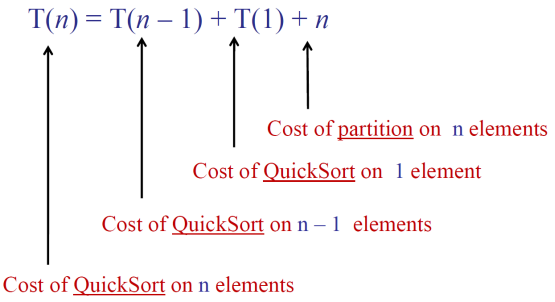
- `Partition` Version 1 (new array): $O(n)$
- `Partition` Version 2 (in-place): $O(1)$
- `QuickSort` (in-place): $O(log(n))$ - due to recursion deferred operations

## 💣Stability

- Stability of Quick Sort depends on partition
- `Partition` Version 1 (new arr) ⇒ Stable, extra memory required
- `Partition` Version 2 (in-place) ⇒ Unstable: due to swaps during partition

## 📍Choice of Pivot

- 1st/Last/Middle element worst case (usually alr sorted)
  - Sorting the array takes n executions of partition.
  - Each call to partition sorts 1 element.
  - Each call to partition of size k takes ≥ k
  - $T(n) = T(n-1) + T(1) + O(n) = n + (n-1) + (n-2) + ... = O(n^2)$




- Median element: $T(n) = 2T(n/2) + O(n) = O(n \cdot log(n))$

$$T(n) = T(n/2) + T(n/2) + n$$

Cost of partition on n elements

Cost of QuickSort on *high* elements

Cost of QuickSort on *low* elements

Cost of QuickSort on n elements

**Randomized Algorithm**

- algorithm makes random choices
  - Average-Case analysis: may be deterministic (output is fully determined by the parameter values and the initial values, not random)
- for every input, there is a good probability of success
- Probability Theory

Flipping an (unfair) coin:
- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

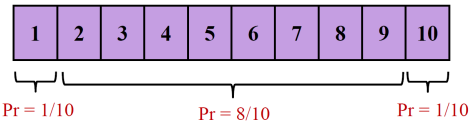
$\mathbf{E}[X] = (p)(1) + (1 - p)(1 + \mathbf{E}[X])$

$= p + 1 - p + 1\mathbf{E}[X] - p\mathbf{E}[X]$

$p\mathbf{E}[X] = 1$

$\mathbf{E}[X] = 1/p$

- A pivot is good if it divides the array into two pieces, each of which is size at least n/10 (arbitrary no.).
  - Expected number of times to repeatedly choose a pivot to achieve a good pivot: `E[# of choices] = 1/p = 10/8 < 2`




Probability of a good pivot:

p = 8/10

(1 – p) = 2/10

$$\mathbf{E}[T(n)] = \mathbf{E}[T(k)] + \mathbf{E}[T(n-k)] + \mathbf{E}[\#\text{ pivot choices}](n)$$
$$<= \mathbf{E}[T(k)] + E[T(n-k)] + 2n$$
$$= O(n \log n)$$

**Choice of Pivot**

| Aa Element | Worst Case | Note |
|---|---|---|
| 1st A[1] | $O(n^2)$ | |
| Last A[n] | $O(n^2)$ | |

| Aa Element | Worst Case | Note |
|---|---|---|
| Middle A[n/2] | $O(n^2)$ | |
| Median | $O(n \cdot log(n))$ | recurrence analysis (assume finding the median takes $O(n)$) $T(n) = T(n/2) + O(n) = O(n)$ Recurse $O(log(n))$ times |
| Random (simplest) | $O(n \cdot log(n))$ | get a good split most of the time $T(n) = O(n)$ Recurse $O(log(n))$ times |

## Count Sort

Condition: input array of known content (e.g. integer from 0 to 255)

1. Create a frequency array (or map if content index is difficult to calculate), loop through the array and increment the frequency.
2. Loop through the input again and modify the array according to the frequency

```
def count_sort(arr : list):
  histogram = [0] * 255
  for i in arr:
    histogram[i] += 1
  cursor = 0
  for i in range(0, len(histogram)):
    for j in range(0, histogram[i]):
      arr[cursor] = i
      cursor += 1
```

## 🤩Optimizations

In practice, more efficient to recurse into smaller half first using `QuickSort` or `MergeSort`.

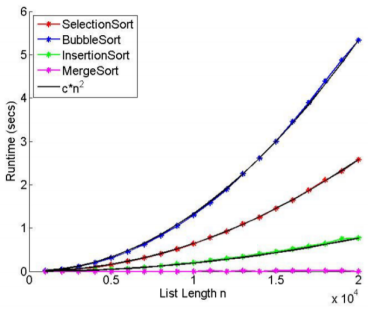
- Less to store on call stack.
- Minimizes depth of call stack to deal with small cases first(?)

**Base Case:**

1. Recurse all the way to single-element arrays.
2. Switch to `InsertionSort` for small arrays.
   - `InsertionSort` is faster on small arrays (n < 1024)
   - `InsertionSort` is faster on nearly sorted arrays
3. Halt recursion early, leaving small arrays unsorted. Then perform `InsertionSort` on entire array.

## 💭Algorithm Comparison Overview

## real world performance




Online: can process without having the entire array available at start

**Comparison**

| Aa Algorithm | Worst | Average | Best | Stability | Loop Invariant | Space Complexity | Note | Online |
|---|---|---|---|---|---|---|---|---|
| Bubble Sort | $O(n^2)$ - inv sorted | $O(n^2)$ | $O(n)$ - sorted | Yes | At the end of iteration j, the biggest j items are correctly sorted in the final j positions of the array. | In-Place<br>O(1) | | |
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | No | At the end of iteration j, the smallest j items are correctly sorted in the first j positions of the array. | In-Place<br>O(1) | | |
| Insertion Sort | $O(n^2)$ - inv sorted | $O(n^2)$ | $O(n)$ - sorted | Yes | At the end of iteration j, the smallest j items are correctly sorted in the first j positions of the array. | In-Place<br>O(1) | | Online |
| Merge Sort | $O(n \cdot log(n))$ | $O(n \cdot log(n))$ | $O(n \cdot log(n))$ | Depends<br>Yes | | O(n) | | |
| Quick Sort | $O(n^2)$ | $O(n \cdot log(n))$ | $O(n \cdot log(n))$ | No | At the end of every loop iteration, `A[high] > pivot`, for all *i* ≥ *high*, `A[i] > pivot`, for all *1* < *j* < *low*, `A[j] < pivot` | Depends<br>In-Place<br>O(1)<br>O(n) | 2-pivot is lil faster | Not Online |
| Heap Sort | $O(n \cdot log(n))$ | | | No | | In-Place<br>O(n) | ternary (3-way) is a lil faster | Not Online |
| Count Sort | $O(n)$ | $O(n)$ | $O(n)$ | No | | O(n) | need to know the content beforehand | Not Online |

**Time Complexity Table**

| Aa Algorithm | Random | Sorted Ascendingly | Sorted Descendingly | Nearly Sorted Asc | Nearly Sorted Des | Homogeneous |
|---|---|---|---|---|---|---|
| (Opt) Bubble Sort | O(n^2) | O(n) | O(n^2) | O(n^2) | O(n^2) | O(n) |
| (Min) Selection Sort | O(n^2) | O(n^2) | O(n^2) | O(n^2) | O(n^2) | O(n^2) |
| Insertion Sort | O(n^2) | O(n) | O(n^2) | O(n) | O(n^2) | O(n) |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n log n) | O(n log n) | O(n log n) |
| Quick Sort (pivot = A[0]) | O(n log n) | O(n^2) | O(n^2) | O(n^2) | O(n^2) | O(n^2) |
| Quick Sort (random pivot) | O(n log n) | O(n log n) | O(n log n) | O(n log n) | O(n log n) | O(n^2) |

# W4L1 Order Statistics

## Quick Select

**Option 1:** Sort the entire array and find `arr[k]`

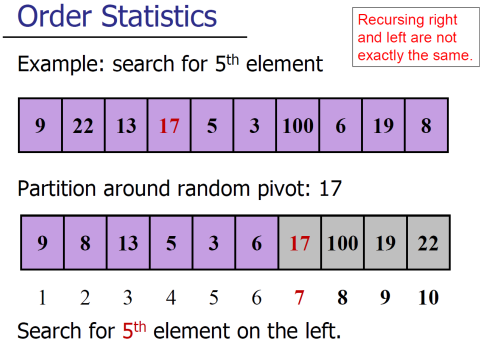
⇒ $T(n) = O(n \cdot log(n))$

**Option 2:** Minimise sorting by **partitioning** the array and continue searching in the correct half ⇒ Only recurse once (in the correct half)

⇒ $T(n) = T(n/2) + O(n) = O(n)$

⇒ O(n) for partition

⇒ starts from 1 not 0




### ⌨️Pseudo-Code

```
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)
```

```
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;
```

# W4L2 Binary Trees

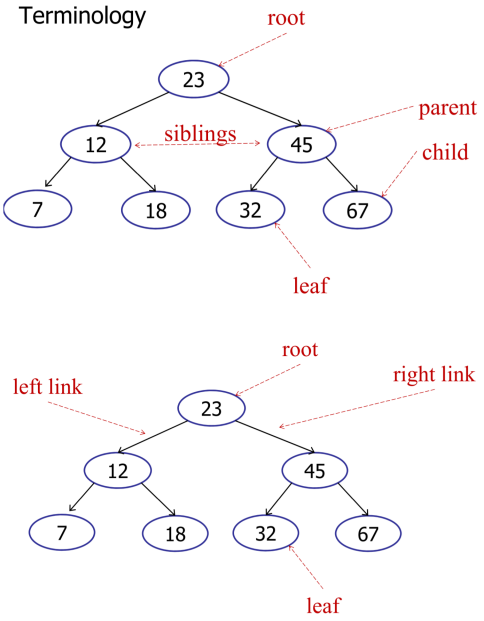
**interface IDictionary**

| Aa Return type | Name | Function | Sorted Arr | Unsorted Arr | Linked List | BST | AVL Tree | Column |
|---|---|---|---|---|---|---|---|---|
| void | `insert (Key k, Value v)` | insert (k, v) into table | add to middle of arr → O(n) | add to end of arr → O(1) | add to head of list → O(1) | O(h) (largest possible = O(n), smallest possible = O(log n)) | O(log n) | |
| Value | `search (Key k)` | get value paired with k | binary search → O(log n) | unsorted → O(n) | list traversal → O(n) | O(h) | O(log n) | |
| Key | `successor (Key k)` | find the next key > k | | | | O(h) | | |
| Key | `predecessor (Key k)` | find the next key < k | | | | O(h) | | |
| void | `delete (Key k)` | remove key k (and corr value) | | | | O(h) | | |
| boolean | `contains (Key k)` | is there a value for k? | | | | | | |
| int | `size ()` | no. of (k, v) pairs | | | | | | |
| int | `findMax` and `findMin` | | | | | O(h) | | |
| Untitled | `inOrderTraversal` | | | | | O(n) | | |
| Untitled | | | | | | | | |

## Terminology & Properties

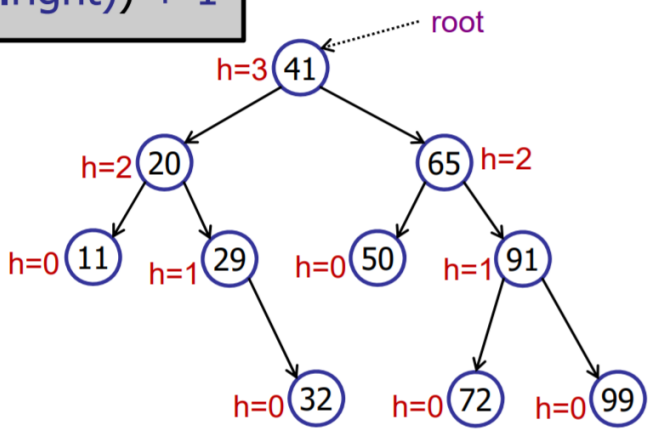
- **BST Property:** all in left sub-tree < key < all in right sub-right
- Two Children: `v.left` and `v.right`
- Key: `v.key`
- Height `h(v)` : Number of edges on longest path from root to leaf.
  - `h(v) = max(h(v.left), h(v.right)) + 1;`
  - `h(null) = -1`
- Shape

- same keys ≠ same shape
- performance depends on shape
- order of insertion determines the shape
- insert keys in a random order ⇒ balanced
- each order does snot yield a unique shape
  - no. of ways to order insertions = n!
  - no. of shapes of a binary tree (Catalan No.) ~ $4^n$
- Cn = # of trees with (n+1) nodes = # expressions with n pairs of matched parentheses




h(v) = 0 (if v is a leaf)

h(v) = max(h(v.left), h(v.right)) + 1




(For simplicity: h(null) = -1)

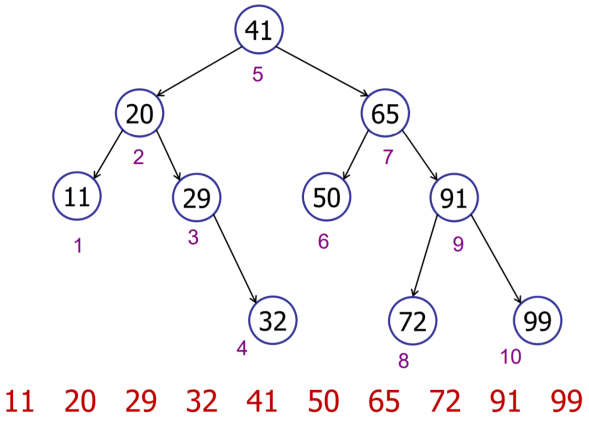
⌨️**Pseudo-Code**

```
/** fields of a Binary Tree **/
public class BinaryTree {
  private BinaryTree leftTree;
  private BinaryTree rightTree;

  private KeyType key;
  private ValueType value;
  // Remainder of binary tree implementation
}

/** Calculate the height of the tree **/
public int height(){
  int leftHeight = -1;
  int rightHeight = -1;
  if (leftTree != null)
    leftHeight = leftTree.height();
  if (rightTree != null)
    rightHeight = rightTree.height();
  return max(leftHeight, rightHeight) + 1;
  // max of subtrees
}

/** Search for the max key **/
public TreeNode searchMax(){
  if (rightTree != null) {
    return rightTree.searchMax();
  }
  else return this; // Key is here!
}

/** Search for the min key **/
public TreeNode searchMin(){
  if (leftTree != null) {
    return leftTree.searchMin();
  }
  else return this; // Key is here!
}

/** Search for a specified key **/
// worst-case T(n) = O(n) = O(height)
public TreeNode search(int queryKey){
  if (queryKey < key) {
    if (leftTree != null)
      return leftTree.search(key);
    else return null;
  }
  else if (queryKey > key) {
    if (rightTree != null)
      return rightTree.search(key);
    else return null;
  }
  else return this; // Key is here!
}

/** Insert a new key **/
public void insert(int insKey, int intValue){
  if (insKey < key) {
    if (leftTree != null)
      leftTree.insert(insKey);
    else leftTree = new TreeNode(insKey,insValue);
  }
  else if (insKey > key) {
    if (rightTree != null)
      rightTree.insert(insKey);
    else rightTree = new TreeNode(insKey,insValue);
  }
  else return; // Key is already in the tree!
}
```

## Tree Traversal: In-Order

- 'dot' at the bottom of the circle

- visits each node at most once
- Sequence:
  1. Left-subtree
  2. *self*
  3. Right-subtree

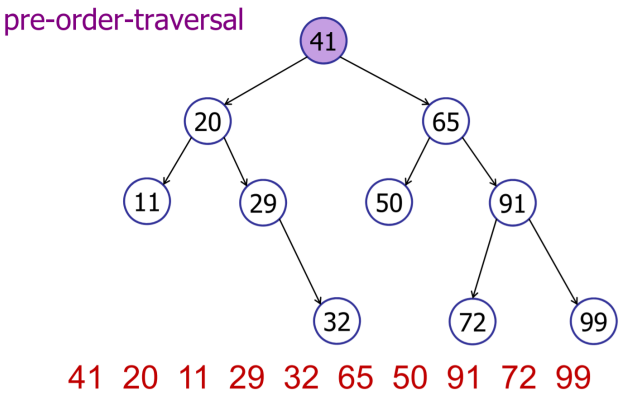



### ⌨️Pseudo-Code

```
public void in-order-traversal(){
  // Traverse left sub-tree
  if (leftTree != null)
    leftTree.in-order-traversal();

  // self
  visit(this);

  // Traverse right sub-tree
  if (rightTree != null)
    rightTree.in-order-traversal();
}

// T(n) = O(n)
// visits each node at most once
// Note: searching for all the items is going to be slower
```

## Tree Traversal: Pre-Order

- 'dot' at the left of the circle
- Sequence:
  1. *self*
  2. Left-subtree
  3. Right-subtree




## ⌨️Pseudo-Code

```
public void pre-order-traversal(){
  // self
  visit(this);

  // Traverse left sub-tree
  if (leftTree != null)
    leftTree.in-order-traversal();

  // Traverse right sub-tree
  if (rightTree != null)
    rightTree.in-order-traversal();
}
```

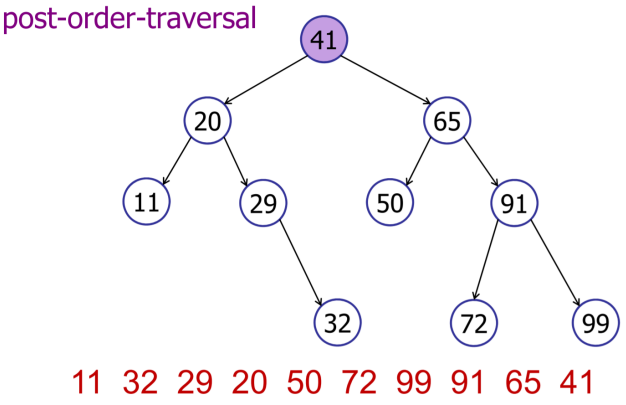
## 🌟Actual Code

```
private void PreOrder(TreeNode node) {
  if(node!=null) {
      System.out.println(node.Value);
      InOrder(node.Left);
      InOrder(node.Right);
  }
}

private void InOrder(TreeNode node) {
  if(Rt!=null) {
      InOrder(node.Left);
      System.out.println(node.Value);
      InOrder(node.Right);
  }
}

------------------------------------------------------------
TreeNode[] enumerateNodes(TreeNode node, Child child) {
    int size = countNodes(node, child);
    TreeNode[] inOrderArr = new TreeNode[size];
    if (child == Child.LEFT) {
        enumerateNodes(node.left, inOrderArr);
    } else {
        enumerateNodes(node.right, inOrderArr);
    }
    index = 0; // reset index after the end of each enumerateNodes call
    return inOrderArr;
}

int index = 0;

void enumerateNodes(TreeNode node, TreeNode[] arr) {
    if (node != null) {
        enumerateNodes(node.left, arr);
        arr[index++] = node;
        enumerateNodes(node.right, arr);
```

```
    }
}
```

## Tree Traversal: Post-Order

- 'dot' at the right of the circle
- Sequence:
  1. Left-subtree
  2. Right-subtree
  3. *self*

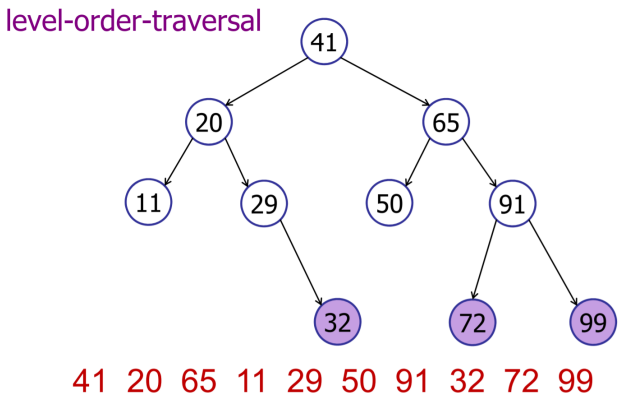



### ⌨️Pseudo-Code

```
public void post-order-traversal(){
  // Traverse left sub-tree
  if (leftTree != null)
    leftTree.in-order-traversal();

  // Traverse right sub-tree
  if (rightTree != null)
    rightTree.in-order-traversal();

  // self
  visit(this);
}
```

## Tree Traversal: Level-Order (BFS)

- Sequence:
  1. root
  2. next level → left → right




## Iterator Interface

Tree implements `Iterable<Key>`

- pre-order iterator
- in-order iterator
- post-order iterator
- level-order iterator

### ⌨️Pseudo-Code

```
private class TreeIterator implements Iterator<Key>{
  BinaryTree currentNode;

  public boolean hasNext(){
    return (current != null);
  }

  public Key next(){
  // What goes here?
  }
}
```

## Successor

- If you search for a key not in the tree → either find predecessor or successor
- successor: the closest element > key
- predecessor: the closest element < key

**Basic Strategy:** `successor (key)`

1. Search for key in the tree
2. If (result > key), then return result
3. If (result ≤ key), then search for successor of result

Case 1: Node has a right child → `sucessor(key) = right.searchMin()`

Case 2: Node has no right child → find in parent node

### ⌨️Pseudo-Code

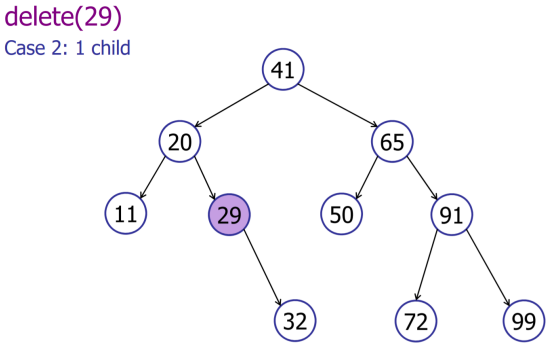
```
public TreeNode successor() {
  if (rightTree != null)
    return rightTree.searchMin();

  TreeNode parent = parentTree;
  TreeNode child = this;

  while ((parent != null) && (child == parent.rightTree))
    child = parent;
    parent = child.parentTree;
  }
  return parent;
}
```

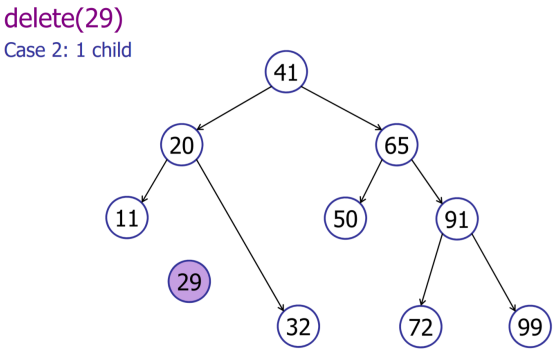
## Delete(v)

- $T(n) = O(height)$
- *refer to Udemy Abstract Class Challenge Part 3 for deletion's actual code*

**Case 1:** no children → just delete `v`

**Case 2:** 1 child → delete `v` → connect with `child(v)` to `parent(v)`



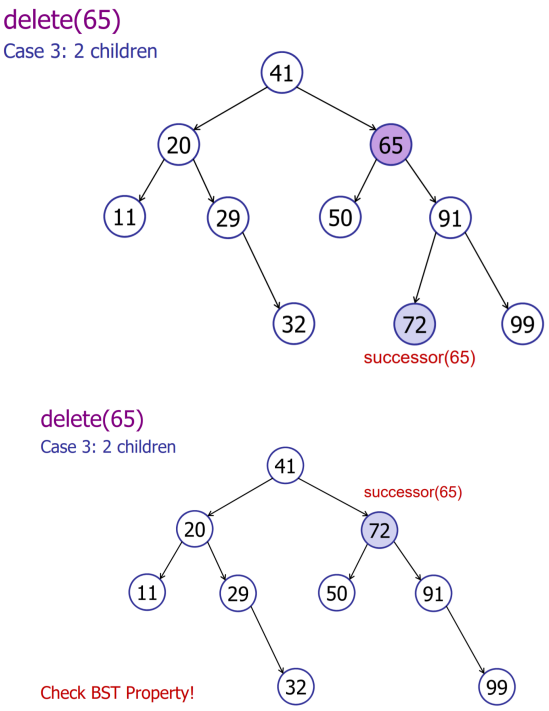




**Case 3:** 2 children → delete the element → reconnect with the successor of the deleted element

- x = `successor(v)`
- delete(x)
- remove `v`
- connect `x` to `left(v)` , `right(v)` , `parent(v)`

Equivalently,

- if v has 2 children
- x = `successor(v)`
- `swap(v, x)`
- delete v from binary tree (and reconnect children)
- for every ancestor of the deleted node:
  - check if it is height-balance
  - if not, perform a rotation, continue until the root
  - **O(log n)** rotations; each rotation costs O(1)

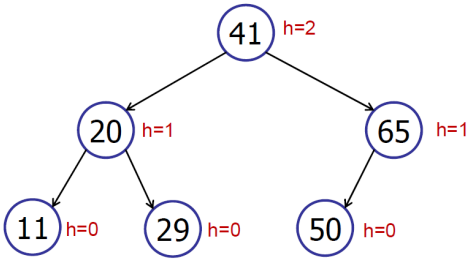



# W5L1 AVL Trees

## Step 0: Augment

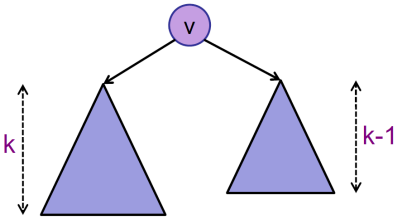
- In every node `v`, store height: `v.height = h(v)`
- Update height for after every `insertion` and `deletion`

```
insert(x)
  if (x < key)
    left.insert(x)
  else right.insert(x)
  height = max(left.height, right.height) + 1 // this line
```



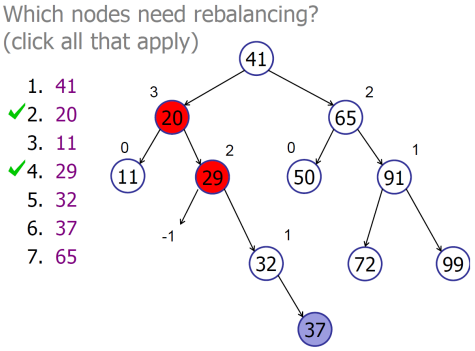

## Step 1: Define Balance Condition (Invariant)

- node `v` is **height-balanced** if: `| v.left.height - v.right.height | <= 1`
- A binary search tree is height balanced if **every node** in the tree is height-balanced.
- A height-balanced tree with n nodes has at most height $h < 2 \cdot log(n)$.
- Equivalently, a height-balanced tree with height h has at least $n > 2^{h/2}$
- log(n) - 1 ≤ h ≤ n
- A height-balanced tree is balanced, i.e. has height $h = O(log(n))$




## Step 2: Maintain Balance

- walk from the bottom up the tree to check for balance at each level → use rotation to rebalance (worst case **2 rotations**)
  - in each case, reduce the height of subtree by 1.
- A is **left-heavy** if the left sub-tree has a larger height than the right subtree.
- A is **right-heavy** if the right sub-tree has larger height than left subtree.

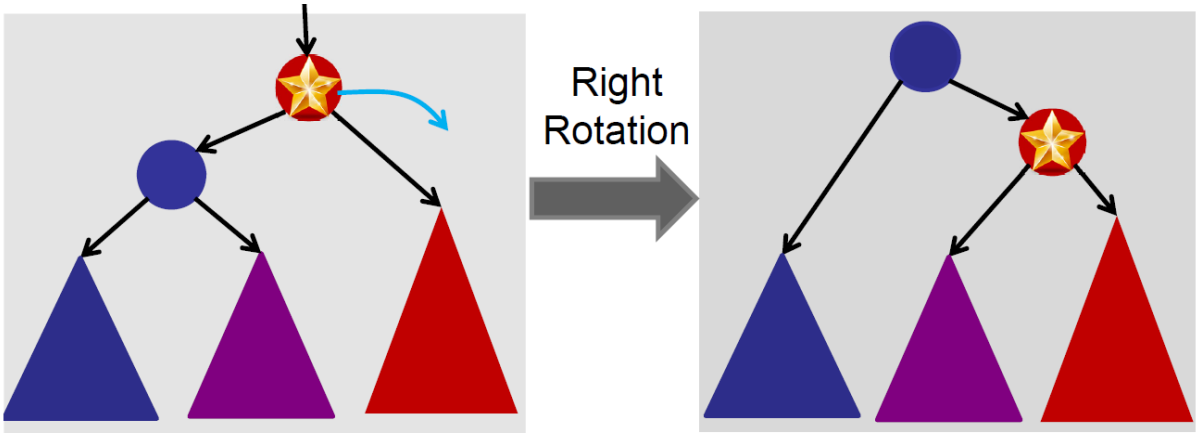



## Tree Rotations - Right (for Left-Heavy)

- Can create every possible tree shape with rotations
- rotations maintain ordering of keys (BST property)

- Left child → Parent
- Parent → Right Child
- Right child of previously left child → left child of previously parent node

- Root of the subtree moves right
- Rotate-right requires a left child




### ⌨️Pseudo-Code

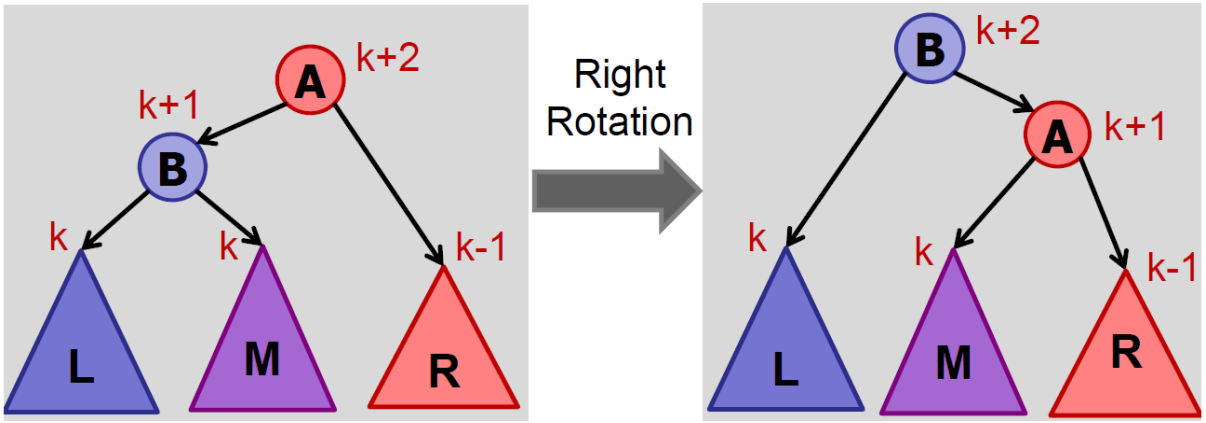
```
right-rotate(v) // assume v has left != null; v is the node
    w = v.left // w is the left child of root node v
    w.parent = v.parent
    v.parent = w
    v.left = w.right
    w.right = v
```

**Assume A is the lowest node in the tree violating balance property.**

**Case 1: A is left-heavy and B is balanced → Right rotation**

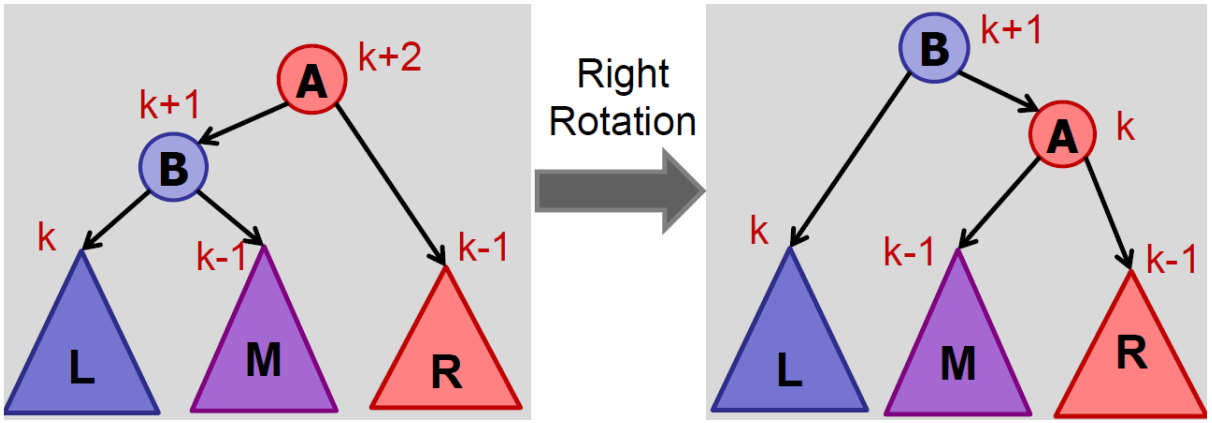
h(L) = h(M)

h(R) = h(B) – 2 = h(M) – 1 = h(L) - 1



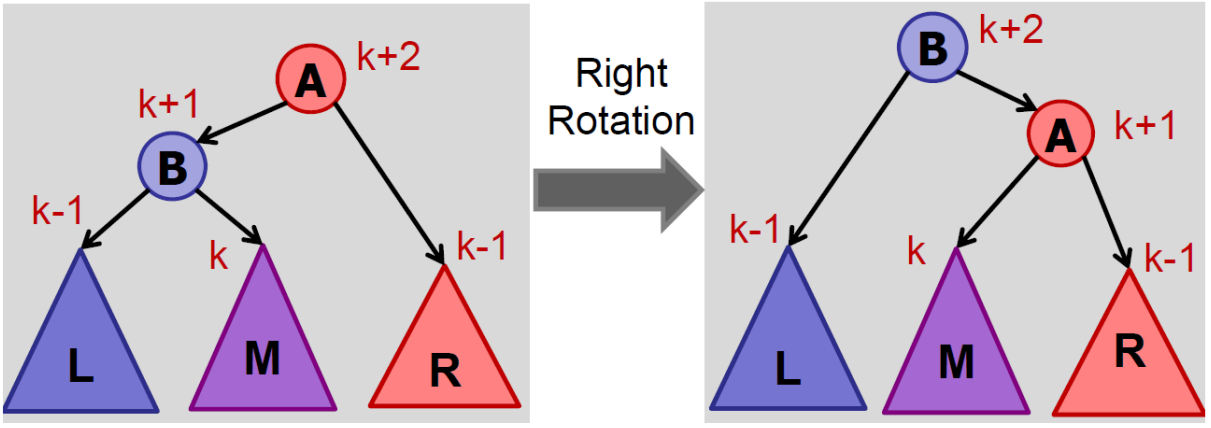

**Case 2: A is left-heavy and B is left-heavy → Right rotation**

h(L) = h(M) + 1 = h(R) + 1

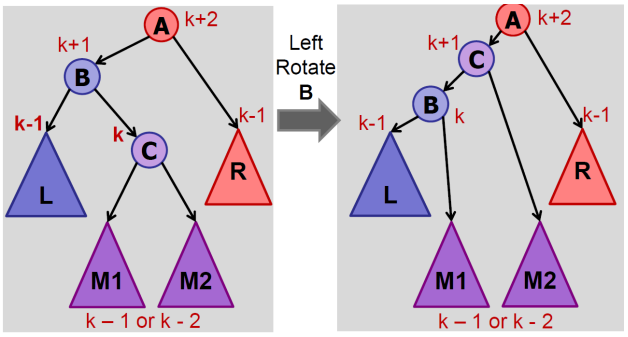



**Case 3: A is left-heavy and B is right-heavy → Right rotation**

h(L) = h(M) - 1 = h(R) - 1

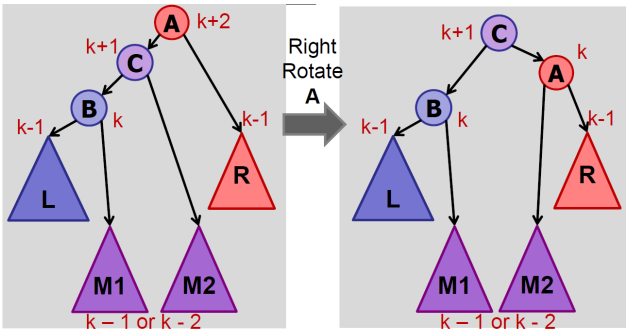



- note: left and right child of root B is still unbalanced
- should left-rotate B first




Left-rotate B

After left-rotate B: **A** and **C** still out of balance.




After right-rotate A: all in balance.

**Summary:**

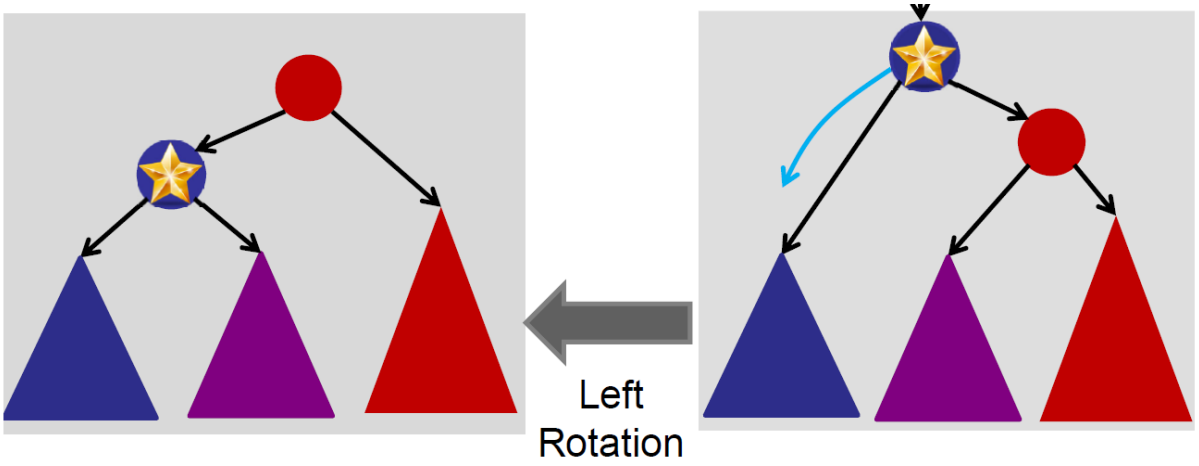
If v is out of balance and left-heavy:

1. `v.left` is balanced: `right-rotate(v)`
2. `v.left` is left-heavy: `right-rotate(v)`
3. `v.left` is right-heavy: `left-rotate(v.left)` and then `right-rotate(v)`

- In the worst case, need **2** rotations after an insertion.
- starting from the bottom-most, rotate in the reverse direction that is heavy
  - e.g. `v` is left-heavy, `v.left` is right-heavy: `left-rotate(v.left)` and then `right-rotate(v)`

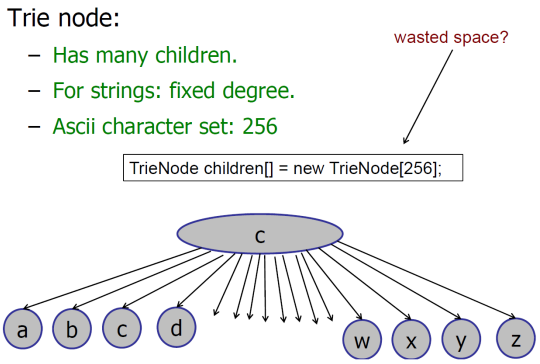
## Tree Rotations - Left (for Right-Heavy)

- root of the subtree moves left
- Rotate-left requires a right child



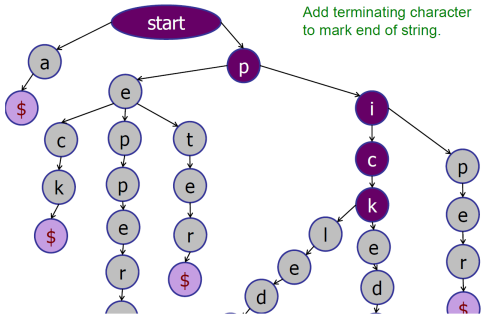

## Trie

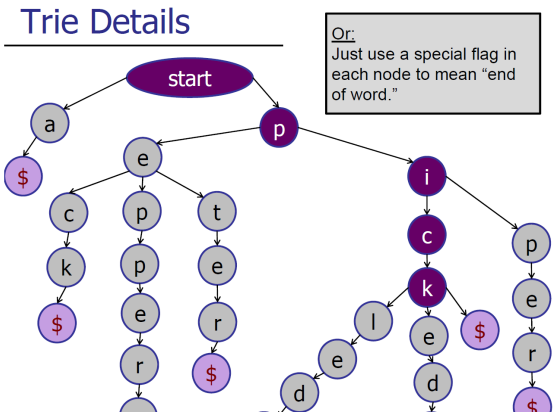
- Compare text strings `min(A.length, B.length)`
- Cost of insertion: **O(hL) = O(L*log n)**
- Cost to search a string of length L: **O(L)**
- Cost for storing a try: **O((size of text)*overhead)**




**Trie Tradeoffs**

| Aa Name | Complexity | Property 2 |
|---|---|---|
| Time | faster: **O(Length of string)** | does not depend on size of total text / no. of strings |
| Space | more space: **O(text size)** | more nodes and more overhead |




# W6L1 Augmentation of bBSTs

## Dynamic Order Statistics

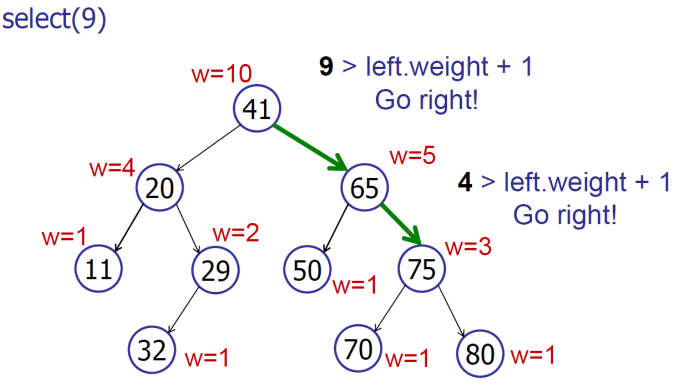
### Solution 1: Sort first before Selection

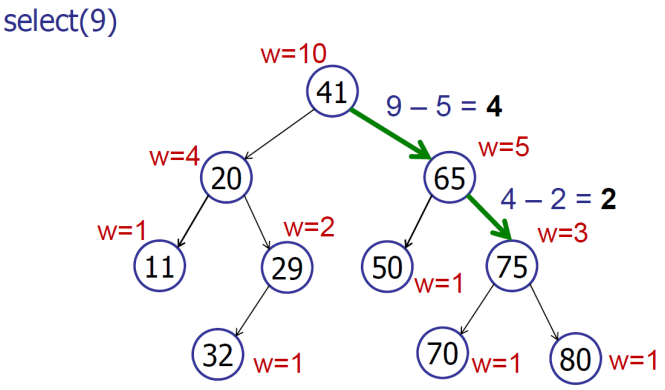
- more efficient when there is more selections than insertions

1. Basic structure: sorted array A
2. `insert(int item)`: add item to sorted array A (compare to determine which index to go, then shift back every element's indexes ⇒ **O(n))**
3. `select(int k)`: return A[k] (in-order traversal) ⇒ **O(1)**
4. `rank(v)`: computes the rank of a node v(key)

**Optimising Selection & Rank**

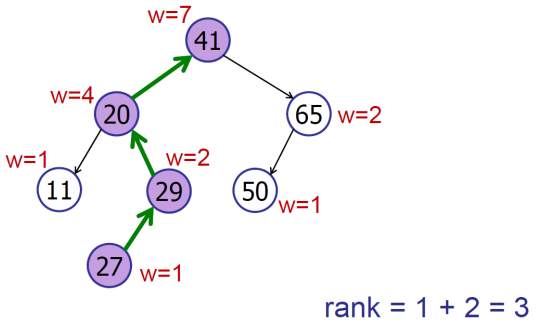
- Idea: store size of sub-tree in every node
- Weight: size of the tree rooted at that node
  - `w(leaf) = 1`
  - `w(v) = w(v.left) + w(v.right) + 1`




- Rank: index of the node in sorted array
  - `rank in subtree = left.weight + 1`

Example: rank(27)



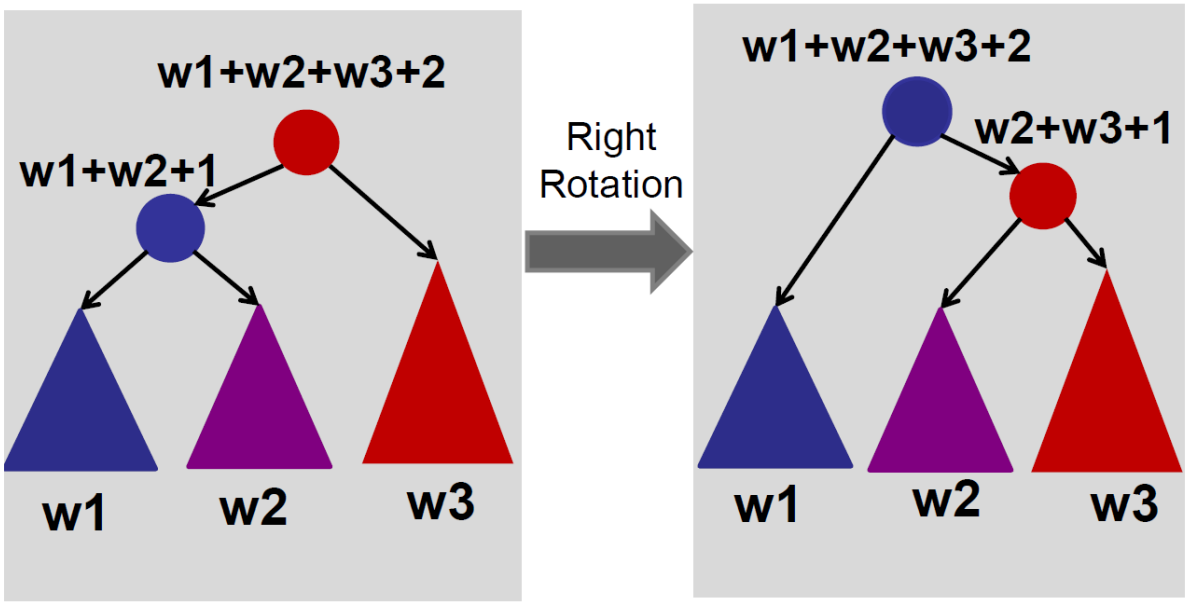

## ⌨️Pseudo-Code for select and rank

```
select(k)
  rank = left.weight + 1; // initial rank
  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); // note (k - rank)

rank(node) // goes from the node to the root
  rank = node.left.weight + 1; // initial rank
  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; // stops when node == root, i.e. node.parent == null
  return rank;
```

**Maintaining info as DS is modified (update weights)**

- insert / delete ⇒ imbalanced ⇒ rotations
- update weights during rotation (arithmetic calculation) ⇒ **T(n) = O(1)**

## Maintain weight during rotations:




### Solution 2: QuickSelect immediately

**Solutions Comparison**

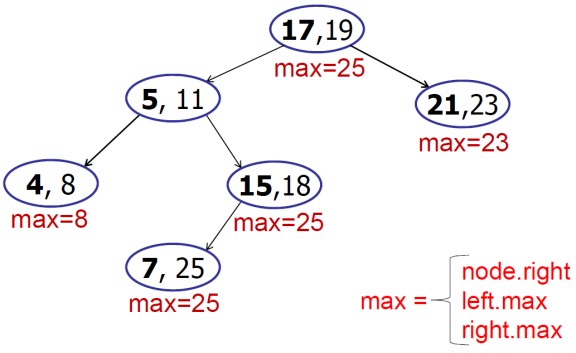
| Aa Solution | Insert | Select |
|---|---|---|
| Solution 1: Sorted Arr | O(n) | O(1) |
| Solution 2: Unsorted Arr | O(1) | O(n) |

## Interval Queries

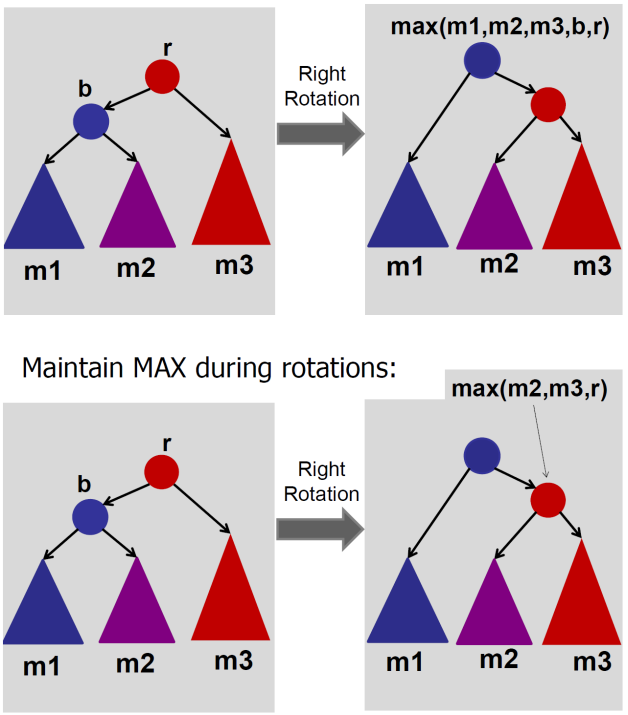
Interval Trees

- Each node is an interval
- Sorted by left endpoint (start point)

Augment: maximum endpoint in subtree



**Maintaining MAX during rotations**




`interval-search(x)`

- find interval containing x
- if search goes right, then there is no overlap in left subtree
  - either search finds key in right subtree or it is not in the tree
- if search goes left and there is no overlap in the left subtree, then it is safe to go left (no overlap in right-subtree & x < every interval in right subtree)
  - either search finds key in left subtree or it is not in the tree
- Conclusion: search finds an overlapping interval, if it exists
- **T(n) = O(log n)**

## ⌨️Pseudo-Code for interval-search(x)

```
interval-search(x) // find interval containing 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 // x is not in the range of any intervals in left subtree
      c = c.right;
    else c = c.left;
  return c.interval;
```

**Extension: List all intervals that overlap with x**

- **T(n) = O(k log n)**
- Best known solution: T(n) = O(k + log n)

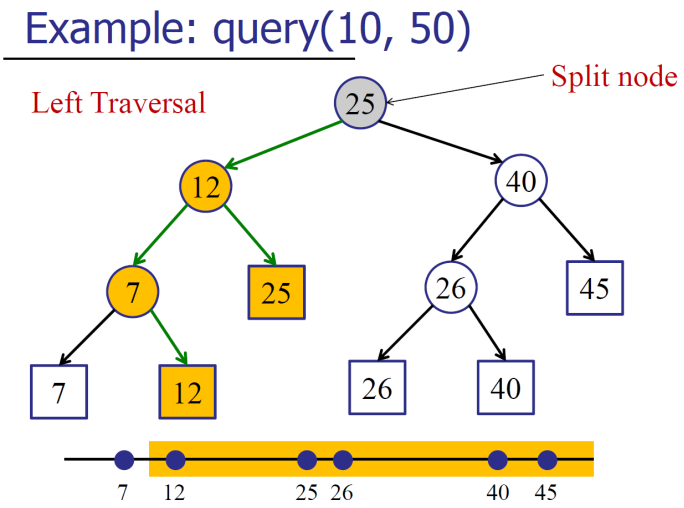
```
All-Overlaps Algorithm:
  Repeat until no more intervals:
    Search for interval.
    Add to list.
    Delete interval.
  Repeat for all intervals on list:
    Add interval back to tree.
```

## Orthogonal Range Searching

**One-Dimension Range Queries**

Strategy:

1. Use a BST
2. Store all points in the leaves of the tree (internal nodes store only copies).
3. Each internal node `v` stores the **MAX** of any leaf in the left sub-tree




Algorithm for Query:

1. Find 'split' node: `v = FindSplit(low, high);`
   - **T(n) = O(log n)**
2. `LeftTraversal(v, low, high);`

   At every step, either:
   - output all right subtree and recurse left, or
   - recurse right
3. `RightTraversal(v, low, high);`

   At every step, we either:
   - output all left subtree and recurse right, or
   - recurse left

### ⌛Time Complexity

- recurse at most **O(log n)** times (else block)
- output all subtrees (if block) ⇒ **O(k)**, where k is the no. of items found

- Overall Query: **T(n) = O(k + log n)**, where k is the no. of items found
- Pre-processing (build tree): **T(n) = O(n log n)**

## 📁Space Complexity

- **S(n) = O(n)**

## ⌨️Pseudo-Code

```
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;

LeftTraversal(v, low, high)
  if (low <= v.key) {
    all-leaf-traversal(v.right); // output all right subtrees (leaves)
    LeftTraversal(v.left, low, high); // recurse left
  }
  else {
    LeftTraversal(v.right, low, high); // recurse right
  }

RightTraversal(v, low, high)
  if (v.key <= high) {
    all-leaf-traversal(v.left); // output all left subtrees (leaves)
    RightTraversal(v.right, low, high); // recurse right
  }
  else {
    RightTraversal(v.left, low, high); // recurse left
  }

/** Version 2: only wants to known the total no. of points in the range **/
LeftTraversal(v, low, high)
  if (low <= v.key) {
    total += v.right.count; // change made
    LeftTraversal(v.left, low, high);
  }
  else {
    LeftTraversal(v.right, low, high);
  }
```

**Two-Dimensional Range Tree**

Input: *n points in  a 2D plane*

Query: *Box*

Problem: cannot enumerate entire subtrees since there may be too many nodes that do not satisfy the y-restriction

Solution (Augment):

- Create a 1d-range-tree on the x-coordinates.
- Each node in the x-tree has a set of points in its subtree.
- Store a y-tree at each x-node containing all the points in the sub-tree using only y-coordinates.

Time Complexity: $T(n) = O(log^2 n + k)$

- O(log n) to find split node.
- O(log n) recursing steps
- O(log n) y-tree-searches of cost O(log n)

- O(k) enumerating output

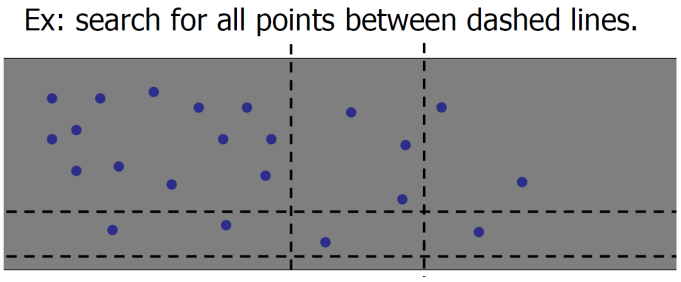
Space Complexity: $S(n) = O(n \cdot log(n))$

- Each point appears in at most one y-tree per level
- There are O(log n) levels ⇒ Each node appears in at most O(log n) y-trees.
- The rest of the x-tree takes O(n) space.

Building the tree: $T(n) = O(n \cdot log(n))$

Insertion / Deletion:

- Every rotation you may have to entirely rebuild the y-trees for the rotated nodes.
- Cost of rotate: O(n)




```
LeftTraversal(v, low, high)
  if (v.key.x >= low.x) {
    ytree.search(low.y, high.y);
    LeftTraversal(v.left, low, high);
  }
  else {
    LeftTraversal(v.right, low, high);
  }
```

**d-dimensional range queries**

- Store d–1 dimensional range-tree in each node of a 1D range-tree.
- Construct the d–1-dimensionsal range-tree recursively.
- Query Time Complexity: $T(n) = O(log^d(n) + k)$
- Build Tree Time Complexity: $T(n) = O(n \cdot log^{d-1}(n))$
- Space Complexity: $S(n) = O(n \cdot log^{d-1}(n))$

**Real World (extra)**

kd-Trees

- Alternative levels in the tree: vertical → horizontal → vertical → horizontal
- Each level divides the points in the plane into half
- Query cost: O(√n) worst-case

# W6L2 Hashing

**Symbol Table**

| Public interface | SymbolTable | Function | Examples | Symbol Table |
|---|---|---|---|---|
| void | `insert(Key k, Value v)` | insert (k, v) into table | Dictionary: key = word, v = definition; Contact list: key = name, v = phone no. Java compiler: key = var name, v = type & value | O(1) |
| Value | `search(Key k)` | get value paired with k | | O(1) |
| void | `delete(Key k)` | remove key k (and value) | | |
| boolean | `contains(Key k)` | check if there is a value for k | | |
| int | `size()` | number of (k, v) pairs | | |
| Untitled | | note: no success/predecessor queries; symbol table is not comparison-based | | |

**Dictionaries v.s. Symbol Tables Sorting**

| Step | Dictionary/AVL | Symbol Table |
|---|---|---|
| Insert | $O(log(n))$ | $O(1)$ |
| Search for min | $O(log(n))$ | $O(1)$ |
| Repeat: find successor | $O(n)$: repeat n times | $O(n^2)$: O(n) to find one successor, total n items |
| Overall | $O(n \cdot log(n))$ | $O(n^2)$ |
| Functionality | Wide interface: lots of functionality (but may be inefficient) | Narrow interface: limited functionality (enforces proper use; restricts usage) |
| Untitled | | |

**java.util.Map**

| Public interface | java.util.Map<Key, Value> | Function | Note |
|---|---|---|---|
| void | `clear()` | remove all entries | |
| boolean | `containsKey(Object k)` | check if k is in the map | |
| boolean | `containsValue(Object v)` | check if v is in the map | may not be efficient |
| Value | `get(Object k)` | get value of k | |
| Value | `put(Key k, Value v)` | adds (k, v) to table | |
| Value | `remove(Object k)` | remove mapping for k | |
| int | `size()` | no. of entries | |
| Set<Map.Entry<Key, Value>> | `entrySet()` | set of all mappings | |
| Set<Key> | `keySet()` | set of all keys | |
| Collection<Value> | `values()` | collection of all values | |
| Untitled | | no successor/predecessor queries | |
| Untitled | | no duplicate keys allowed | |
| Untitled | | no mutable keys: if an object is used as a key, that object cannot be modified later | |

| Aa Public interface | java.util.Map<Key, Value> | Function | Note |
|---|---|---|---|
| Untitled | | unsorted | |
| Untitled | | not necessarily efficient to work with the available sets/collections | |
| Untitled | | | |

## Hash Functions

- Hashing is an improvement over Direct Access Table.
- The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table
- In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table for faster access of elements.
- The efficiency of mapping depends of the efficiency of the hash function used.
- e.g. `h(k) = k % 10` , `h(k) = k mod 5`

A good hash function should have following properties

1. Efficiently computable.
2. Should uniformly distribute the keys (Each table position equally likely for each key)

Hash Table

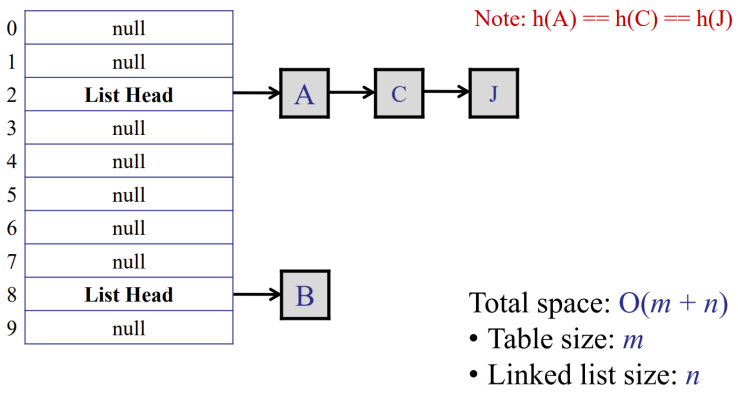
- An array that stores pointers to `Value` (e.g. records) corresponding to a given `Key` (e.g. phone number)
- An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry
- Note: should not use mutable keys in hash table → search may not work when initial key is changed

#### Collisions

- The situation where a newly inserted key maps to an already occupied slot in hash table
- 2 distinct keys k1 and k2 collide if $h(k1) = h(k2)$
- unavoidable: table size < universe size (by PHP, there will always more than one element in one slot/bucket)

### Separate Chaining

- Idea: make each cell of hash table point to a linked list of records that have same hash function value.
  - put both (collision) items in the same bucket.
  - each bucket contains a linked list of items (m buckets, n entries)
- Chaining is simple, but requires additional memory outside the table.




**Simple Uniform Hashing**

Assumptions

- Every key is equally likely to map to every bucket
  - should not randomize → search cost becomes very slow
- keys are mapped independently
- No. of items/entries = n
- No. of buckets = m
- **load(hash table) = average # items per bucket = n/m**
- Expected search time = 1 (due to hash function + array access) + expected # items per bucket (due to linked list traversal)
- **Expected search time = 1 + n/m = O(1)**
  - E[A+B] = E[A] + E[B]
  - `X(i, j) = 1` if item `i` is put in bucket `j`
  - `X(i, j) = 0` if item `i` is NOT put in bucket `j`
  - $P(X(i,j) == 1) = 1/m$
  - $E(X(i,j)) = P(X(i,j) == 1) \cdot 1 + P(X(i,j) == 0) \cdot 0 = 1/m$
  - **E(no. of items per bucket) = n/m**
  - e.g. expected search runtime when 1000 elements are inserted into a hash table with chaining with 20 buckets = 1 + 1000/20 = 51
- if m = n, then  with high probability, every linked list has length $O(log \cdot n/log(log \cdot n))$, and so each search operation terminates in time $O(log \cdot n/log(log \cdot n))$ with high probability
- Expected max cost for inserting n elements = $O(log \cdot n/log(log \cdot n))$

**Operations:**

`insert(key, value)`

- *allows duplicates; preventing duplicates requires searching → slower runtime*
- calculate `h(key)`
- lookup `h(key)` and
- add `(key, value)` to the linked list ⇒ **O(1)**
- $T(n) = O(1 + cost(h)) = O(1)$

`search(key)`

- calculate `h(key)`
- search for `(key, value)` in the linked list ⇒ O(n) in worst case (all keys hash to the same bucket)
- **Expected search time = O(1)**
- **Worst-case search time = O(n)**
- $T(n) = O(n + cost(h))$

### Rules for `hashCode()`

1. always returns the same value, if the object has not changed
2. if 2 objects are equal, then they return the same `hashCode`.
   - it is legal but not useful for every object to return a same constant value
3. must override `equals()` to be consistent with `hashCode()`

### Default Java Implementation for `hashCode`

- `hashCode` returns  the memory location of the object
- every object hashes to a different location ⇒ must override `hashCode()` for every class for hash table to work
- `hashCode` is always a 32-bit integer
- every 32-bit integer gets a unique `hashCode`

```
/*
>> (Signed right shift):
All integers are signed in Java.
If the number is negative, then 1 is used as a filler.
If the number is positive, then 0 is used as a filler.

>>> (Unsigned right shift):
It always fills 0 irrespective of the sign of the number.
*/
System.out.println(Integer.toBinaryString(-1));
// prints "11111111111111111111111111111111"
System.out.println(Integer.toBinaryString(-1 >> 16));
// prints "11111111111111111111111111111111"
System.out.println(Integer.toBinaryString(-1 >>> 16));
// prints "1111111111111111"
// = 00000000000000001111111111111111

System.out.println(Integer.toBinaryString(121));
// prints "1111001"
System.out.println(Integer.toBinaryString(121 >> 1));
// prints "111100"
System.out.println(Integer.toBinaryString(121 >>> 1));
// prints "111100"
```

```
// hashCode() implementation for Long
public int hashCode() {
  return (int) (value ^ (value >>> 32));
}

// hashCode implementation for String
public int hashCode() {
  int h = hash; // only calculate hash once
  if (h == 0 && count > 0) { // empty = 0
    int off = offset;
    char val[] = value;
    int len = count;
```

```
    for (int i = 0; i < len; i++) {
      h = 31*h + val[off++];
    }
    hash = h;
  }
  return h;
}
/*
hash =  s[0]*31^(n-1) +
        s[1]*31^(n-2) +
        s[2]*31^(n-3) +
        ... +
        s[n-2]*31 +
        s[n-1]
*/
```

**Rules for `equals()`**

- Reflexive: `x.equals(x) == true`
- Symmetric: `x.equals(y) == y.equals(x)`
- Transitive: `x.equals(y), y.equals(z)` ⇒ `x.equals(z)`
- Consistent: always returns the same answer
- Null is null: `x.equals(null) == false` (null == null)

```
// Overriding equals for Pair
boolean equals(Object p){
  if (p == null) return false;
  if (p == this) return true;

  if (!(p instanceOf Pair)) return false;
  Pair pair = (Pair)p;

  if (pair.first != first) return false;
  if (pair.second != second) return false;

  return true;
}
```

```
// HashMap
public V get(Object key) {
  if (key == null) return getForNullKey();
  int hash = hash(key.hashCode());
  for (Entry<K,V> e = table[indexFor(hash,table.length)]; e != null; e = e.next) {
    Object k;
    if (e.hash == hash &&((k = e.key) ==key) || key.equals(k)))
// assignment of k to e.key
    return e.value;
  }
  return null;
}
```

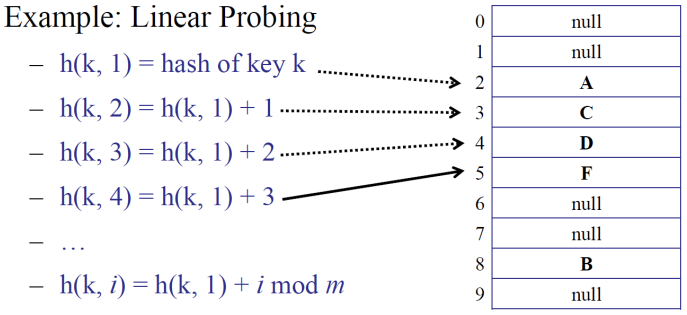
## Open Addressing

- idea: all elements are stored in the hash table itself. Each table entry contains either a record or `null`.
- When searching for an element, we examine table slots one by one until the desired element is found or it is clear that the element is not in the table.
- A hash collision is resolved by probing (i.e. searching through alternate locations in the array)
- Adv:

1. no linked list
2. all data are stored directly in the table
3. one item per slot

**Linear Probing**

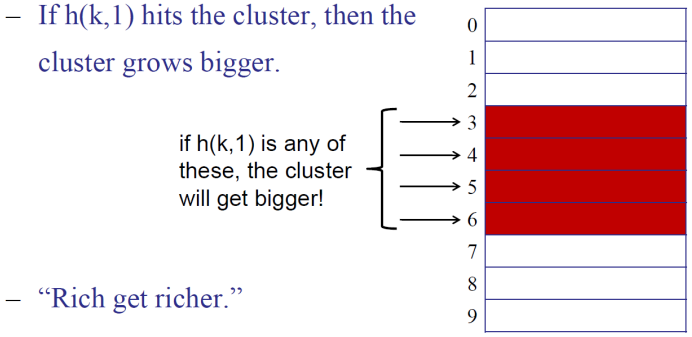
- linearly probe for next slot
- redefine hash function: `h(key, i); U → {1 ... m}`
  - `key`: the item to map
  - `i`: no. of collisions




```
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
```

Problems:

- not considered a good hash function (does not satisfy Simple Uniform Hash Assumption)
- Clustering:
  - Primary: if there is a cluster, then there is a higher probability that the next `h(k)` will hit the cluster ⇒ runtime slows as more time is taken to find a free slot or to search an element
    - if the table is 1/4 full, the there will be clusters of size O(log n)
  - Secondary: 2 record will have the same collision chain (probe sequence) if their initial position is the same




In practice:

- linear probing is very fast
- cheap to access nearby array cells

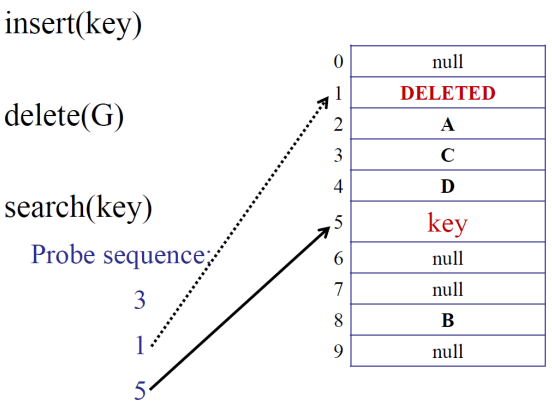
- cache may hold the entire cluster → slowing down of search time is not severe

### ⌨️ Pseudo-Code for hash-insert, hash-search

```
// insert: Keep probing until an empty slot is found.
// Once an empty slot is found, insert key.
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(); // Table full!

// search: Keep probing until slot's key doesn't become equal to k
// or an empty slot is reached.
hash-search(key)
  int i = 1;
  while (i <= m) {
    int bucket = h(key, i);
    if (T[bucket] == null) // Empty bucket!
      return key-not-found;
    if (T[bucket].key == key) // bucket has been filled with key
      return T[bucket].data;
    i++;
  }
return key-not-found; // Exhausted entire table.
```

`Delete(key)`:

- If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted". (if probe reaches `null`, the search will just return `key-not-found`)
- If `insert(key)` finds `DELETED`, overwrite the deleted cell




**Properties of a good hash function**

1. `h(key, i)` **enumerates all possible buckets**
   - For every bucket `j`, there is some `i` such that `h(key, i) = j`
   - the hash function is permutation of `{1...m}`

**2. Simple Uniform Hashing Assumption**

- Every key is equally likely to be mapped to every bucket (permutation), independently of every other key
- $n!$ permutations for probe sequence

    - if the sequence is not a permutation, `insert` returns `table-full` even when there is still space left
- linear probing satisfies this
    - e.g. 1234; 1243; 1423...
    - max n probes
    - random buckets: max O(n log n) probes (probability of getting the same bucket)
    - Linear probing does NOT satisfy this: only go by sequential order i.e. 1234 or 2345...

**Double Hashing:**

- start with 2 ordinary hash functions: `f(k)`, `g(k)`
- define new hash function: `h(k, i) = f(k) + i*g(k) mod m`

```
let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S
```

**Operation Performance**

- Load factor (how full the table is) α = n/m (assume that this < 1)
- Expected time to insert/search/delete ≤ 1/(1 - α)
- e.g. if (α = 90%), then E[# probes] = 10

**Separate Chaining v.s. Open Addressing**

| Aa Name | E(search time) | m == n | Space usage | Sensitivity to clustering & load | Cache performance & Memory allocation | Usefulness | Implementation |
|---|---|---|---|---|---|---|---|
| Separate Chaining | E(insert) = O(1) ≤ worst: O(n); // with Simple Uniform Hashing Assumption Amortized(insert) = O(1) E(search/delete) = O(1+n/m) ≤ worst: O(n) (no amortized for search since no resizing) | Hash table never fills up, we can always add more elements to chain → can still search efficiently | wastage of space (some are not used) | less sensitive to the hash function or load factors | cache performance is not good as keys are storing using linked lists (extra space) | mostly used the no. of items to be inserted is unknown | Simpler |
| Open Addressig | E(insert/search/delete) = O(1/(1 - α)) = O(1/(1-n/m)) | Table is full → cannot insert any more items → cannot search efficiently | slot can be used even if an input does not map to it | - requires extra care to avoid clustering and load factor (performance degrades badly as α → 1) - try to keep α ≤ 0.75 | better cache performance as everything is stored in the same table (in one place in memory) | used when the no. of items is known | More computation |
| Untitled | | | | | | | |

## Table Size

- Assume hashing with chaining & simple uniform hashing
- **E(search time) = O(1+n/m)**
- **Optimal size: m = θ(n)**
  - m < 2n: too many collisions
  - m > 10n: too much wasted space

Resizing Table:

1. Choose new table size `m`
2. Choose new hash function `h`
   - hash function depends on table size
   - `h : U → {1...m}`
3. For each item in the old hash table
   - compute new hash function
   - copy item to new bucket

### ⌛Time Complexity for resizing table

- Assume
  - `m1`: size of old hash table (<n)
  - `m2`: size of new hash table (>2n)
  - `n`: no. of elements in the hash table
- Costs:
  - scanning old hash table: `O(m1)`
  - insert each element in new hash table: `O(1)`
  - initializing a new table: `O(m2)`
  - Total: $O(m_1 + m_2 + n) = O(n)$

Idea 1: Increment table size by 1

– When ($n == m$): $m = m+1$

– Cost of each resize: O($n$)

| Table size | 8 | 8 | 9 | 10 | 11 | 12 | … | n+1 |
|---|---|---|---|---|---|---|---|---|
| Number of items | 0 | 7 | 8 | 9 | 10 | 11 | … | n |
| Number of inserts | | 7 | 1 | 1 | 1 | 1 | … | 1 |
| Cost | | 7 | 8 | 9 | 10 | 11 | | n |

– Total cost: $(7 + 8 + 9 + 10 + 11 + \ldots + n) = O(n^2)$

Idea 2: Double table size

- When ($n == m$): $m = 2m$
- Cost of each resize: O($n$)

| Table size | 8 | 8 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 32 | 32 | 32 | … | 2n |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # of items | 0 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | … | n |
| # of inserts | | 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | … | 1 |
| Cost | | 7 | 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 16 | 1 | 1 | | n |

- Total cost: $(7 + 15 + 31 + \ldots + n) = O(n)$

Most insertions = O(1); some O(n) ⇒ avg = O(1)

| Table size | Total Resizing Cost |
|---|---|
| 8 | 8 |
| 16 | (8 + 16) |
| 32 | (8 + 16 + 32) |
| 64 | (8 + 16 + 32 + 64) |
| 128 | (8 + 16 + 32 + 64 + 128) |
| … | … |
| m | <(1+2+4+8+…+m) ≤ O(m) |

Idea 3: Square table size

- When ($n == m$): $m = m^2$

| # Items | Total Resizing Cost |
|---|---|
| 8 | 64 |
| 64 | (64 + 4,096) |
| 4,096 | (64 + 4,096 + …) |
| … | … |
| $n$ | > $n^2$ |
| | = O($n^2$) |

O(m1+m2+n)=O(n^2 + n); inefficient space usage

**Deleting Elements**

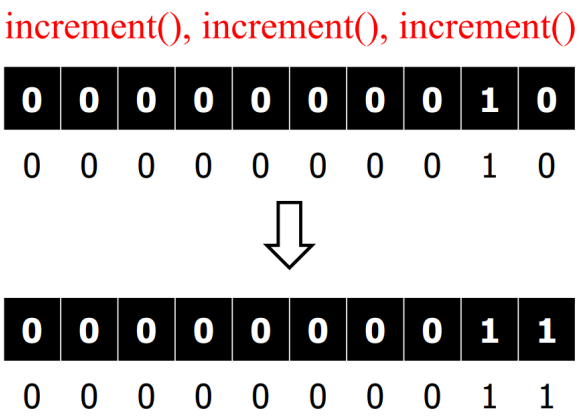
- if `n == m` then `m = 2m`
  - Every time you double a table of size `m`, at least `m/2` new items were added.
- if `n < m/4` then `m = m/2`
  - Every time you shrink a table of size `m`, at least `m/4` items were deleted

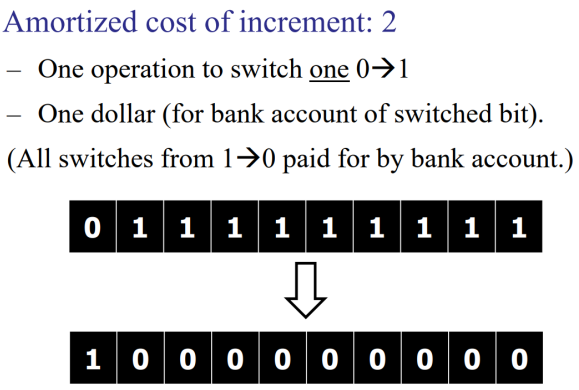
## Amortized Analysis

- Definition: operation has amortized cost $T(n)$ if for every integer k, cost of k operations ≤ $k \cdot T(n)$
- technique for calculating the average cost (but does NOT mean average)

- e.g. inserting k elements into a hash table takes O(k) time ⇒ amortize cost of insertion = O(1)
- Accounting method:
  - each table has a bank account
  - each time an element is added to the table, it adds O(1) dollars to the bank account
  - resizing a table of size m: at least m/2 new elements since last resize ⇒ bank acc has θ(m) dollars
  - Inserting k elements cost:
    - deferred dollars: O(k) - to pay for resizing ⇒ amortized: O(1)
    - immediate dollars: O(k) - for inserting elements ⇒ amortized: O(1)
    - total: O(k) ⇒ O(1)/operation

Binary Counter




- Worst-case cost of incrementing a counter with max-value n = O(log n)

**Designing Hash Functions**

- Goal: find a hash function whose values look random

**Division Method:** `h(k) = k mod m`

- 2 keys collide when `k1 = k2 mod m`
- collision is unlikely if keys are random
- choose m = prime no.
  - not too close to a power of 2 or 10
- division is slow

**Multiplication method**

- Fix table size: $m = 2^r$, for some constant r
- Fix word size: w, size of a key in bits
- constant A: odd integer $> 2^{w-1}$
  - if A is even → lose ≥ 1 bit's worth
  - big enough: use all the bits in A
- faster than division

– Assume $k$ and $m$ have common divisor $d$.

$k \bmod m + i*m = k$

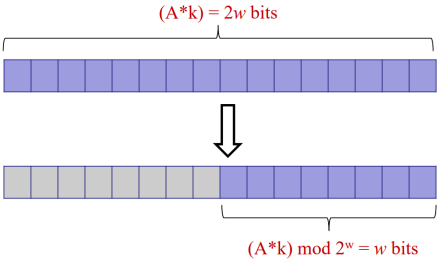
divisible by $d$

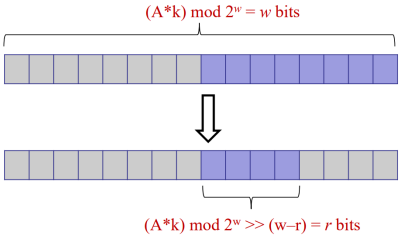
– Implies that h($k$) is divisible by $d$.

– If all keys are divisible by $d$, then you only use 1 out of every $d$ slots

| | |
|---|---|
| 0 | A |
| 1 | null |
| 2 | null |
| d = 3 | B |
| 4 | null |
| 5 | null |
| 2d =6 | C |
| 7 | null |
| 8 | null |
| 3d =9 | D |

– Given m, w, r, A: $h(k) = (Ak) \bmod 2^w \gg (w - r)$



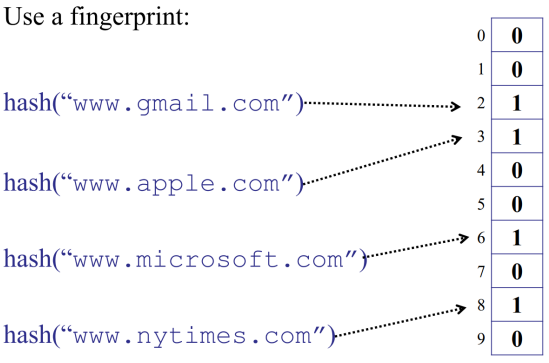

# W8L2 Set

## Fingerprints Hash Table(FHT - Set ADT)

**Copy of Set**

| Aa public class | Set<Key> | Purpose | T(n) |
|---|---|---|---|
| void | `insert(Key k)` | insert `k` into a set | O(k), k no. of hash functions |
| boolean | `query(Key k)` | check if `k` is in the set | O(k) |
| void | `delete(Key k)` | remove `k` from the set | O(k) |
| void | `intersect(Set<Key> s)` | take the intersection: - bitwise AND of 2 Bloom filters | O(m) |
| void | `union(Set<Key> s)` | take the union - bitwise OR of 2 Bloom filters | O(m) |
| Untitled | | | |

- maintain a vector of 0/1 bits ⇒ reduced space
- FHT does not store the key in the table




- No false negatives: if key is in the table, it will always return true

- P(false negatives) = 0

- False positives present: if key is not in the table, it may still return true (due to collision)

On lookup in a table of size m with n elements,
Probability of **no** false positive:

$$\left(1-\frac{1}{m}\right)^n \approx \left(\frac{1}{e}\right)^{n/m}$$

chance of no collision

Probability of **no** false positive: (simple uniform hashing assumption)

$$\left(1-\frac{1}{m}\right)^n \approx \left(\frac{1}{e}\right)^{n/m}$$

Probability of a false positive, at most:

$$1-\left(\frac{1}{e}\right)^{n/m}$$

– Probability of false positives $< p$

- Example: at most 5% of queries return false positive.

$p = .05$

– Need: $\frac{n}{m} \leq \log\left(\frac{1}{1-p}\right)$

- usually has a bigger table to avoid collisions ⇒ increased space

Bloom Filter uses ≥2 hash functions to further avoid collisions & minimise false positives

Probability a given bit is 0:

$$\left(1-\frac{1}{m}\right)^{2n} \approx \left(\frac{1}{e}\right)^{2n/m}$$

Probability of a false positive: (1 set in both slots)

$$\left(1-\left(\frac{1}{e}\right)^{2n/m}\right)^2$$

* Assuming BOGUS fact that each table slot is independent…

Assume you want:

- probability of false positives $< p$
  - Example: at most 5% of queries return false positive.

    $p = .05$
- Need: $\frac{n}{m} \leq \frac{1}{2}\log\left(\frac{1}{1-p^{1/2}}\right)$

- k hash functions

Probability a given bit is 0:

$$\left(1-\frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

Probability of a collision at one spot:

$$1-e^{-kn/m}$$

* Assuming BOGUS fact that each table slot is independent…

Probability of a collision at one spot:

$$1-e^{-kn/m}$$

Probability of a collision at all $k$ spots:

$$\left(1-e^{-kn/m}\right)^{k}$$

Choose: $k = \frac{m}{n}\ln 2$

Error probability: $2^{-k}$

- deleting an element
  - store counter instead of 1 bit
  - on insertion: increment
  - on deletion: decrement
  - note:
    - if counter is big ⇒ space-expensive;
    - if collisions are rare, counter is small ⇒ only a few bits

| | | | |
|---|---|---|---|
| 0 | 0 | & | 0 |
| 1 | 0 | & | 1 |
| 2 | 0 | & | 0 |
| 3 | 1 | & | 1 |
| 4 | 0 | & | 0 |
| 5 | 0 | & | 0 |
| 6 | 1 | & | 0 |
| 7 | 0 | & | 0 |
| 8 | 1 | & | 1 |
| 9 | 0 | & | 0 |

intersection

- when to use Bloom filters:
  - storing a set of data
  - space complexity is important
  - false positives are ok
- trade-offs: space, time, error probability.

## ⌨️ Pseudo-Code for insert & query

```
insert(key)
  h = hash(key)
  table[h] = 1;

// version 1: 1 hash function
query(key)
  h = hash(key);
  return (table[h] == 1);

// version 2: 2 hash functions (Bloom filter)
query(key)
  k1 = hash1(key);
  k2 = hash2(key);
  return (table[k1] == 1) && (table[k2] == 1)
```

**AVL v.s. Trie**

| Aa Name | T(n): insert, delete, search | S(n) |
|---|---|---|
| AVL | O(L log N) | O(total_string_length) |
| Trie | O(L) | O(total_string_length): more overhead cost |
| Untitled | | |