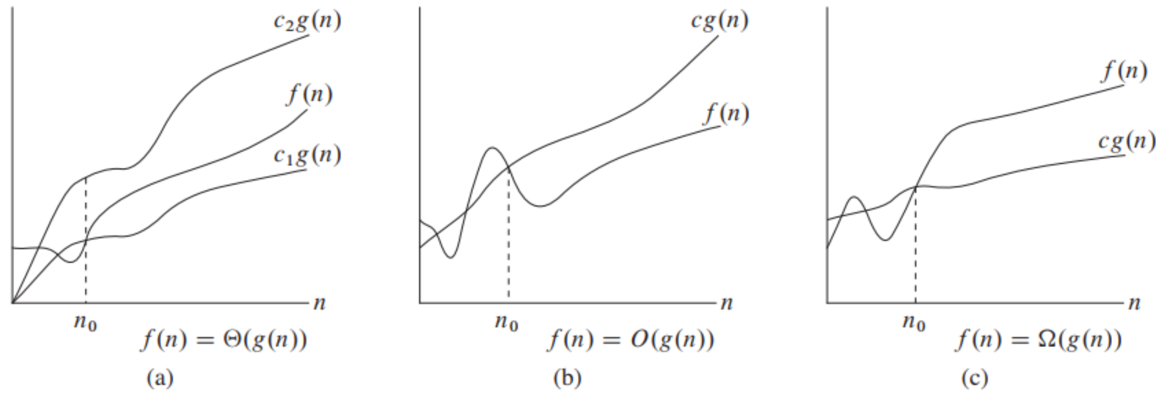
Asymptotic Notation

- **asymptotic notation** is often used to describe the running times of algorithms
- it is a way of abstracting the actual function that describes the running time of an algorithm



Θ-notation *"asymptotically tight bound"*

- $\Theta(g(n))$ denotes the set of functions $f(n)$
- a function $f(n)$ belongs to the set if there exists positive constants c_1, c_2 such that it can be "sandwiched" between $c_1g(n)$ and $c_2g(n)$ for sufficiently large n
  - "sufficiently large n" can be expressed as "for all $n \geq n_0$"
- we often abuse notation and say "a function $f(n)$ = $\Theta(g(n))$" or "... is $\Theta(g(n))$"
  - we mean $\in \Theta(g(n))$
  - but the abuse is useful because if we write something like $2n^2 + \Theta(n)$ it is clear we mean $2n^2 + f(n)$ where $f(n) \in \Theta(n)$

O-notation *"asymptotic upper bound"*

Ω-notation *"asymptotic lower bound"*

"$\Theta = O + \Omega$" Theorem

- for any 2 functions $f(n), g(n)$ we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Insertion Sort

Code

```
function insertionSort(A: number[]) {
    for(let j = 1; j < A.length; j++) {
        const key = A[j];
        // insert A[j] into the sorted sequence A.slice(0, j - 1)
        let i = j - 1;
        while (i >= 0 && A[i] > key) {
            A[i + 1] = A[i];
            i--;
        }
        A[i + 1] = key;
    }
    return A;
}
```

Design

- uses an **incremental** approach
    - having sorted the subarray `A.slice(0, j - 1)`, we insert the single element `A[j]` into its proper place

Correctness

- refer to the code
- notice that `j` holds the "current card" being sorted into the "hand"
- we state properties of subarray `A.slice(0, j - 1)` as a **loop invariant**
    - the subarray `A.slice(0, j - 1)` are elements *originally* in positions 0 through $j - 1$ but in sorted order
- to understand why an algorithm is correct, we must show 3 things about a loop invariant
    1. **initialization**: it is true prior to first iteration of the loop
    2. **maintenance**: if it is true before an iteration of the loop, it remains true before the next iteration
    3. **termination**: when the loop terminates, the invariant gives us a useful property that helps show the algorithm is correct
- applying this to `insertionSort`
    1. **initialization**: `j = 1` before the first iteration such that `A.slice(0, j - 1)` consists of a single element `A[0]` such that the subarray is trivially sorted
    2. **maintenance**: (informally) the body of the for loop moves `A[j - 1]`, `A[j - 2]`, `A[j - 3]`, and so on by one position to the right until it finds the proper position for `A[j]` such that `A.slice(0, j)` will hold elements originally in positions 0 through j but in sorted order. Thus, incrementing `j` for the next iteration of the for loop preserves the loop invariant

3. **termination**: the for loop terminates when `j == A.length == n` such that `A.slice(0, n)` must be sorted at termination

Runtime Analysis

Counting Approach for Iterative Algorithms

- on approach for *iterative algorithms* is to count the number of times each statement is executed
- define constants c for the execution time of each statement
- finally, develop a function describing runtime as a function of the problem size n

line number	cost	times	comments
2	c_2	n	The for loop condition check runs n times.
3	c_3	$n-1$	Its body breaks before the last check.
5	c_5	$n-1$	
6	c_6	$\sum_{j=2}^{n} t_j$	t_j is the number of times the while loop test is executed for `j` (i.e. at most 1 the first time, 2 the second, etc...)
7	c_7	$\sum_{j=2}^{n}(t_j-1)$	
8	c_8	$\sum_{j=2}^{n}(t_j-1)$	
19	c_{10}	$n-1$	

Developing a Function

$$T(n) = c_2 n + c_3(n-1) + c_5(n-1) + c_6 \sum_{j=2}^{n} t_j$$

$$+c_7 \sum_{j=2}^{n}(t_j-1) + c_8 \sum_{j=2}^{n}(t_j-1) + c_{10}(n-1)$$

Best Case

- in the *best case*, the array sorted and $t_j = 1$ for all j such that
  - $\sum_{j=2}^{n} t_j = (n-1)\cdot 1$
  - $\sum_{j=2}^{n}(t_j-1) = 0$

$$\begin{aligned} T(n) &= c_2 n + c_3(n-1) + c_5(n-1) + c_6(n-1) + c_{10}(n-1) \\ &= an + b \end{aligned}$$

In the best case, we have **linear growth function**.

Worst Case

- in the *worst case*, the array is in reverse order such that the $t_j = j$ for all j
- recall that the sum of an arithmetic series is $S_n = \frac{n}{2} \cdot (a_1 + a_n)$
  - where n is the number of terms in the series
  - a_1 is the first term
  - a_n is the last term
- thus
  - $\sum_{j=2}^{n} t_j = \sum j = \frac{n-1}{2}(2+n) = (n-1)(n+2)/2$
  - $\sum_{j=2}^{n}(t_j - 1) = \sum(j-1) = \frac{n-1}{2}(1+n-1) = n(n-1)/2$

$$
\begin{gathered}
T(n) = c_2 n + c_3(n-1) + c_5(n-1) \\
+c_6(n-1)(n+2)/2 + c_7 n(n-1)/2 + c_8 n(n-1)/2 + c_{10}(n-1) \\
= an^2 + bn + c
\end{gathered}
$$

In the worst case, we have **quadratic polynomial function**.

Average Case

In the average case, $t_j = j/2$ for all j and it will also yield a quadratic polynomial function.

Merge Sort

Code

`merge`

```
export function merge(A: number[], start: number, middle: number, end: number) {
    // create 2 arrays with extra slot
    const L = new Array(middle - start + 1);
    const R = new Array(end - middle + 1);
    // copy elements to subarrays
    for (let l = 0; l < L.length - 1; l++) L[l] = A[start + l];
    for (let r = 0; r < R.length - 1; r++) R[r] = A[middle + r];
    // fill extra slot with sentinal
    L[L.length - 1] = Number.MAX_VALUE;
    R[R.length - 1] = Number.MAX_VALUE;
    let l = 0;
    let r = 0;
    // compare elements to order original array
    for (let i = start; i < end; i++) {
        if (L[l] < R[r]) {
            A[i] = L[l];
            l++;
        } else {
            A[i] = R[r];
            r++;
        }
    }
}
```

`mergeSort`

```
export function mergeSort(A: number[], start: number, end: number) {
    if (start < end - 1) {
        let middle = Math.floor((start + end) / 2);
        mergeSort(A, start, middle);
        mergeSort(A, middle, end);
        merge(A, start, middle, end);
    }
}
```

Design

- uses a divide-and-conquer approach which are usually **recursive** in structure

Runtime Analysis

TODO

Solving Recurrences

Substitution Method

- comprises of 2 steps:
  1. guess the form of the solution
  2. use mathematical induction to find the constants and show that the solution works
- we substitute the guessed solution for the function when applying the inductive hypothesis to smaller values

Example

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$$
$$T(1) = 1$$

- *guess* that the solution is $T(n) = O(n \lg n)$[1]
- induction requires us to show that $T(n) \le cn \lg n$ for an appropriate choice of the constant $c > 0$
- assume that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$
- this yields $T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)$[2]
- substitute the above [2] back into the original

$$\begin{aligned} T(n) &\le 2(c\lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)) + n \\ &\le cn \lg (n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\le cn \lg n && \text{if } c \ge 1 \end{aligned}$$

- induction also requires us to show the solution holds for the boundary conditions
  - recall [asymptotic notation]() requires us to prove for "sufficiently large n" or $n \ge n_0$ where we get to choose what n_0 is
  - the base case $T(1) = 1 \not\le c(1) \lg (1) = 0$ goes against out hypothesis
  - notice that for any $n > 3$, our relation does not depend on $T(1)$
  - this leaves us with $n = 2$ and $n = 3$ that we must prove works with our hypothesis

$$\begin{aligned} T(2) &= 2T(1) + 2 = 4 \\ &\le c(2) \lg (2) = 2c \\ T(3) &= 2T(1) + 3 = 5 \\ &\le c(3) \lg (3) \approx 4.75c \end{aligned}$$

  - lastly, we complete the proof $T(n) \le cn \lg n$ by choosing $c \ge 2$

Subtracting lower-order from the guess

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

- *guess* that the solution is $T(n) = O(n)$
- we must show $T(n) \le cn$[1] for some choice of c

- assuming the boundary holds for all $m < n$, in particular $m = \lfloor n/2 \rfloor$ and $m = \lceil n/2 \rceil$
- this yields $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor$ and $T(\lceil n/2 \rceil) \leq c\lceil n/2 \rceil$ [2]
- substituting the above [2] back into the original

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 \end{aligned}$$

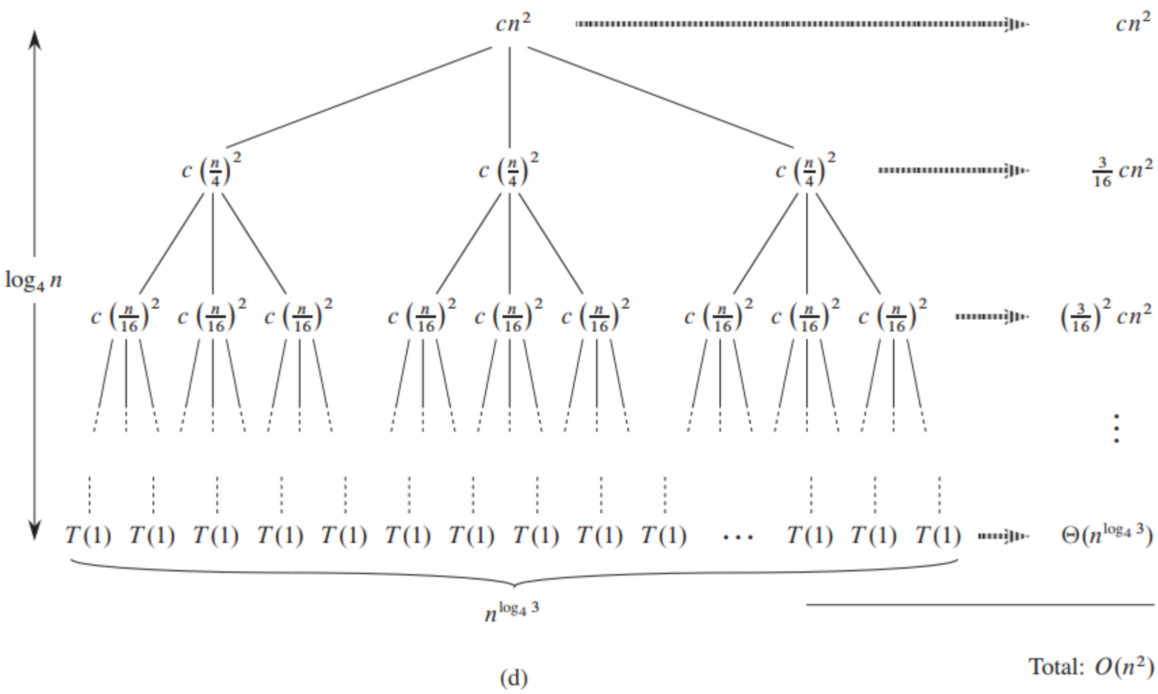
- however, this doesn't imply $T(n) \leq cn$ for any c
- we can solve this by *strengthening* our hypothesis to $T(n) \leq cn - d$ [3]
- again, assuming the boundary holds for all $m < n$, in particular $m = \lfloor n/2 \rfloor$ and $m = \lceil n/2 \rceil$
- this yields $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor - d$ and $T(\lceil n/2 \rceil) \leq c\lceil n/2 \rceil - d$ [4]
- substituting the above [4] back into the original

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \\ &\leq cn - d && \text{if } d \geq 1 \end{aligned}$$

Recursion Tree

Example 1

Solve the upper bound of $T(n) = 3T(\frac{n}{4}) + cn^2$.



- how far from the root do we reach 1?
  - the subproblem for a node at depth i is $\frac{n}{4^i}$
  - the subproblem size is 1 when $n(\frac{1}{4})^i = 1$ or when $i = \log_4 n$
  - thus, the tree has $\log_4 n + 1$ levels (at depths $0, 1, 2, \ldots, \log_4 n$)
- what is the cost at each level of the tree excluding the final level?

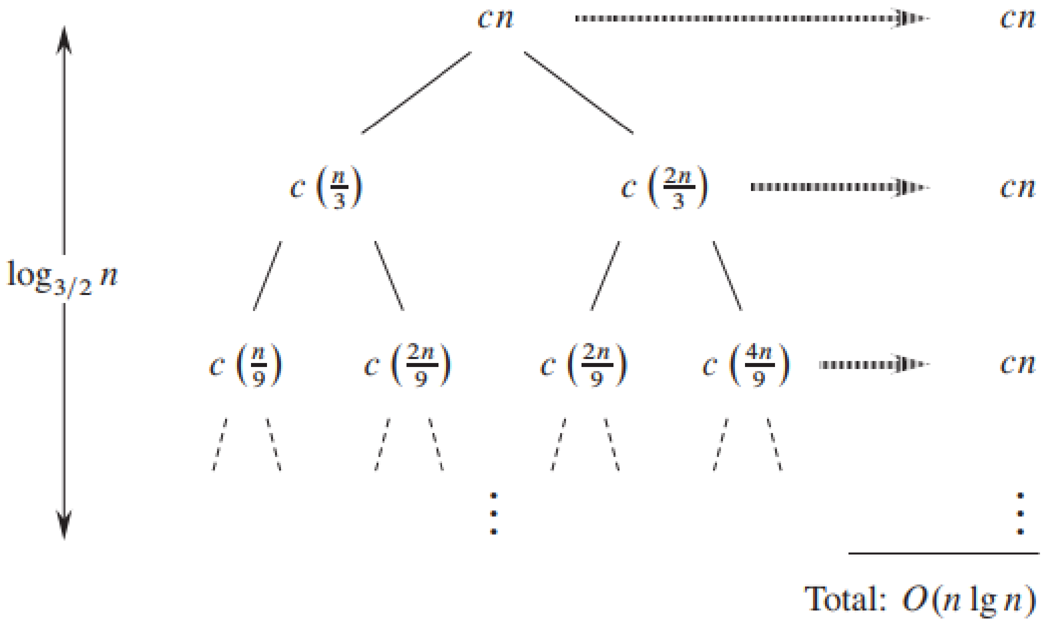
- each level has 3 times more nodes than the one before so the number of nodes at depth i is 3^i
- the subproblem size reduces by a factor of 4 for each level we go down, each node at depth i for $i = 0, 1, 2, \ldots, \log_4 n - 1$ (i.e. all the levels but the last) has a cost of $c(\frac{n}{4^i})^2$ such that the total cost at depth i is $3^i c(\frac{n}{4^i})^2 = (\frac{3}{16})^i cn^2$

- what is the cost at the final level of the tree?
  - the final level at depth $\log_4 n$ has $3^{\log_4 n} = n^{\log_4 3}$ nodes
  - each node has a cost $T(1)$ which we can assume is a constant
  - thus, the final level of the tree has a cost of $\Theta(n^{\log_4 3})$
- what is the sum of all the costs?

$$\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \ldots + (\frac{3}{16})^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&= k * cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)
\end{aligned}$$

Example 2

Solve the upper bound for $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + cn$.



- how far from the root do we reach 1?
  - notice that the tree can't be balanced as some branch paths will reach further to get to 1
  - the longer branch path will be that of $T(\frac{2n}{3})$
  - using this longer branch path, the subproblem size is 1 when $(\frac{2}{3})^i n = 1$ or when $i = \log_{\frac{3}{2}} n$

$$\begin{aligned}
i &= \log_{\frac{3}{2}} n \\
&= \frac{\lg n}{\lg \frac{3}{2}} \\
&= c \lg n \\
&= O(\lg n)
\end{aligned}$$

- what is the cost at each level of the tree?

    - notice at every level, it is cn
- what is the sum of all costs?
    - if we pretend the tree is balanced (which is fine for our upper bound since we'll be overestimating)
    - there are $O(\lg n)$ levels
    - each level is cn
    - thus, $T(n) \leq O(\lg n) * cn = O(n \lg n)$

Master Theorem

Applicable for recurrence relations in the form of:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

> Note that master theorem only solves *some* case.

Cases

Case 1

- if $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$,
- then $T(n) = \Theta(n^{\log_b a})$.

Case 2

- if $f(n) = \Theta(n^{\log_b a})$,
- then $T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3

- if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and
- if $af(\frac{n}{b}) \leq cf(n)$ for some constant $c > 1$ and sufficiently large n,
- then $T(n) = \Theta(f(n))$

Example 1

- solve the asymptotic bound of $T(n) = T(\frac{2n}{3}) + 1$
- $a = 1, b = \frac{3}{2}, f(n) = 1$
- $\log_b a = \log_{\frac{3}{2}} 1 = 0$
- we can easily show the tight bound by $f(n) = 1 = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_b a})$
- this is case 2 such that $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\lg n)$

Example 2

- solve the asymptotic bound of $T(n) = 3T(\frac{n}{4}) + n \lg n$
- $a = 3, b = 4, f(n) = n \lg n$
- $\log_b a = \log_4 3 = 0.8$
- we can show the lower bound by $f(n) = n \lg n = \Omega(n^{0.8+0.2}) = \Omega(n^{\log_b a + \epsilon})$

- where $\epsilon = 0.2$
- this is case 3 such that $T(n) = \Theta(f(n)) = \Theta(n \lg n)$

Example 3

- solve the asymptotic bound of $T(n) = 2T(\frac{n}{2}) + n^2$
- $a = 2, b = 2, f(n) = n^2$
- $\log_b a = \log_2 2 = 1$
- we can show the lower bound by $f(n) = n^2 = \Omega(n^{1+1}) = \Omega(n^{\log_b a + \epsilon})$
  - where $\epsilon = 1$
- this is case 3 such that $T(n) = \Theta(f(n)) = \Theta(n^2)$

Extended Form of Master Theorem

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Extended Form Cases

Case 1

- if $af(\frac{n}{b}) = cf(n)$ is true for some constant $c < 1$,
- then $T(n) = \Theta(f(n))$

Case 2

- if $af(\frac{n}{b}) = cf(n)$ is true for some constant $c > 1$,
- then $T(n) = \Theta(n^{\log_b a})$

Case 3

- if $af(\frac{n}{b}) = f(n)$ is true,
- then $T(n) = \Theta(f(n) \log_b n)$

Methodology

1. list values of $a, b, f(n)$
2. plug a, b in to evaluate $af(\frac{n}{b})$
3. set $af(\frac{n}{b}) = cf(n)$ and solve for c
4. match the value of c to the above cases

Example

- solve the asymptotic bound of $T(n) = 3T(\frac{n}{2}) + n$
- $a = 3, b = 2, f(n) = n$

$$af(\frac{n}{b}) = cf(n)$$
$$\frac{3}{2}n = cn$$
$$c = \frac{3}{2}$$
$$c > 1$$

- this is <u>case 2</u> such that $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\lg 3})$

Selection Sort

Code

```
function selectionSort(A: number[]) {
    for (let i = 0; i < A.length; i++) {
        // find index of minimum element
        let min = i;
        for (let j = i; j < A.length; j++) {
            if (A[j] < A[min]) min = j;
        }
        // swap with front of sorted subarray
        if (min != i) {
            const tmp = A[i];
            A[i] = A[min];
            A[min] = tmp;
        }
    }
    return A;
}
```

Design

- the algorithm maintains 2 subarrays in a given array
- in every iteration, the minimum element from the unsorted subarray is added to the end of the sorted subarray

Runtime Analysis

The first iteration will run $n-1$ times, the second will run $n-2$, and so one until 1.

$$\begin{aligned}(n-1)+(n-2)+\ldots+1 &= \sum_{i=1}^{n-1}(n-i) \\ &= \frac{(n-1)((n-1)+1)}{2} \\ &= \Theta(n^2)\end{aligned}$$

- note that you can use the sum of an arithmetic series formula to show this
- also notice that the runtime analysis is *always* $\theta(n^2)$ for best, worst, and average cases

Bubble Sort

Code

```
function bubbleSort(A: number[]) {
    for (let i = A.length; i > 0; i--) {
        let noSwap = true;
        // compare adjacent elements
        // bubbles float to surface
        for (let j = 0; j < i - 1; j++) {
            if (A[j] > A[j + 1]) {
                noSwap = false;
                const tmp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = tmp;
            }
        }
        if (noSwap) break;
    }
    return A;
}
```

Design

- repeatedly step through the array, compare adjacent elements, and swap if they are in the wrong order
- repeat until list is sorted which is confirmed by *no swaps* occurring in the iteration

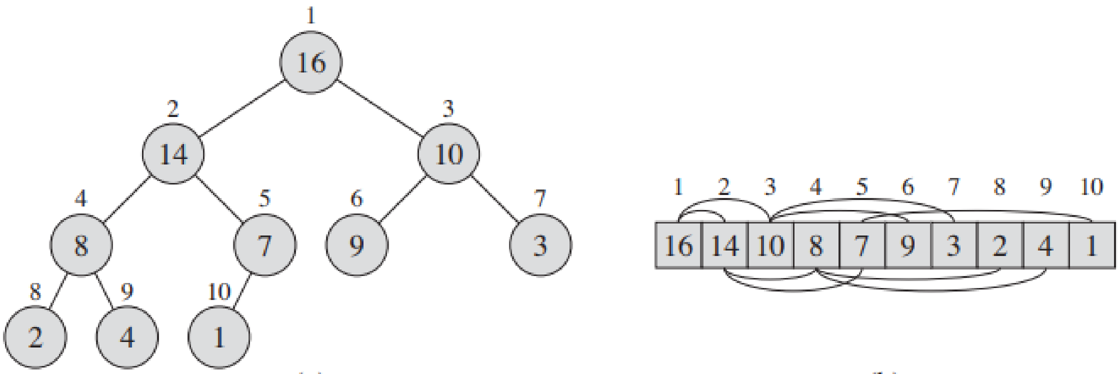
Runtime Analysis

- worst case is $O(n^2)$
- best case is $O(n)$ if we terminate early after an iteration of no swaps
- average case is $O(n^2)$ (using expectation)

Heap Sort

Heaps 🧺

Binary Heap



(a) (b)

- a binary heap is an almost complete binary tree that is stored in an array
  - we will use a variable `size` to determine which parts of the array constitue the heap
- given index `i`, the following can be used to calculate a node's parent and left / right children

```
const parent = (i: number) => Math.floor(i / 2) + 1;
const left = (i: number) => 2 * i + 1;
const right = (i: number) => 2 * i + 2;
```

Max Heap Property

For any node i (excluding the root), $A[\text{parent}(i)] \geq A[i]$.

Code

`maxHeapify`

```
function maxHeapify(A: number[], i: number, size: number) {
    const l = left(i);
    const r = right(i);
    let largest: number;
    // find the largest tree
    if (l < size && A[l] > A[i]) largest = l;
    else largest = i;
    if (r < size && A[r] > A[largest]) largest = r;
    if (largest != i) {
        // swap
        swap(A, i, largest);
        // recurse
        maxHeapify(A, largest, size);
    }
}
```

- `maxHeapify` assumes that the trees at `left(i)` and `right(i)` are already max heaps, but the property is violiated at `i`
- it finds the largest of the two children and swaps with the parent `i`. Then it recurses down the child branch it swapped with
- it terminates when `i` is already the largest

`buildMaxHeap`

```
function buildMaxHeap(A: number[]) {
    for (let i = parent(A.length - 1); i >= 0; i--) {
        maxHeapify(A, i, A.length);
    }
    return A.length;
}
```

- takes an array `A` and builds a max heap out of it
- it returns the `size` of the heap
  - which is just `A.length` since a max heap was built using the entire array

`heapSort`

```
function heapSort(A: number[]) {
    let size = buildMaxHeap(A);
    for (let i = A.length - 1; i > 0; i--) {
        swap(A, 0, i);
        size--;
        maxHeapify(A, 0, size);
    }
}
```

- starting from the lastmost element, we swap it with the root of the tree
  - where the root was previously the largest element the tree as a result of the max heap property
- next, we decrement the size of the heap as the element last swapped with is the next largest element
- finally we call `maxHeapify` to correct for the swapped element in the root
  - notice that the root's left and right children are *still* max heaps as we excluded the largest element from our heap using `size--`

Correctness

Runtime Analysis

Max Heapify

- children subtrees have a size of at most $\frac{2n}{3}$ such that we can describe the recurrence relation as

$$T(n) \leq T(\frac{2n}{3}) + \Theta(1)$$

- we can solve this using the master thereom

$$a = 1, b = \frac{3}{2}, f(n) = \Theta(1)$$
$$\log_a a = \log_{\frac{3}{2}} 1 = 0$$
$$f(n) = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_b a})$$

- this is case 2 such that $T(n) = \Theta(n^0 \lg n) = \Theta(\lg n)$
- alternatively, you can characterize the runtime on the height of a binary tree which would be $\lg n$

Build Max Heap

- each call to `maxHeapify` is $O(\lg n)$
- n of these calls are made such that the upper bound is $O(n \lg n)$
- *we can show a tight bound but I don't feel like it* 🙄

Heap Sort

- if we put the previous stuff together we get

$$\begin{aligned} T(n) &= O(n \lg n) + O(n \lg n) + O(1) \\ &= O(n \lg n) \end{aligned}$$

- the nice thing about heap sort is that we can sort in place meaning only a constant number of array elements are stored outside the input array
    - notice in merge sort we do not have this feature

Quicksort

Code

```
function quicksort(A: number[], p: number, r: number) {
    if (p < r - 1) {
        const q = partition(A, p, r);
        quicksort(A, p, q);
        quicksort(A, q + 1, r);
    }
}

function partition(A: number[], p: number, r: number): number {
    const x = A[r - 1];
    let i = p - 1;
    for (let j = p; j < r - 1; j++) {
        if (A[j] < x) {
            i++;
            swap(A, i , j);
        }
    }
    swap(A, i + 1, r - 1);
    // return the pivot's index
    return i + 1;
}
```

Design

- pick one element as the pivot from the array
  - in our case, it is `A[r]` the last element of the array
- partition the array into 2 subarrays
  - where all elements in the left subarray are less than or equal to the pivot
  - and all elements in the right subarray are greater than or equal to the pivot
- in both subarrays, recursively partition them
- notice `pivot` sorts in place so no extra space is needed

Runtime Analysis

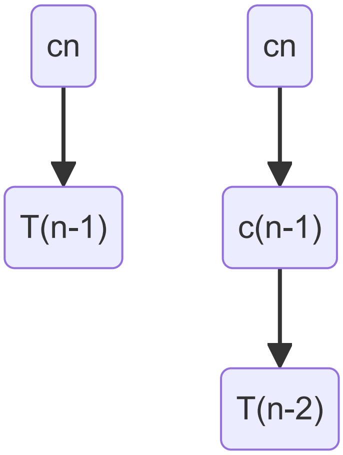
$$T(n) = T(a) + T(b) + \Theta(n)$$

- where $\Theta(n)$ is the complexity of `partition`
- a is the elements in the left subarray and b is the elements in the right subarray after partition finishes

Worst Case

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \Theta(n) \\
&= T(n-1) + cn
\end{aligned}
$$

The worst case partition is that we have $n-1$ elements in the left subarray but 0 in the right (meaning we happened to pick the largest element as our pivot).

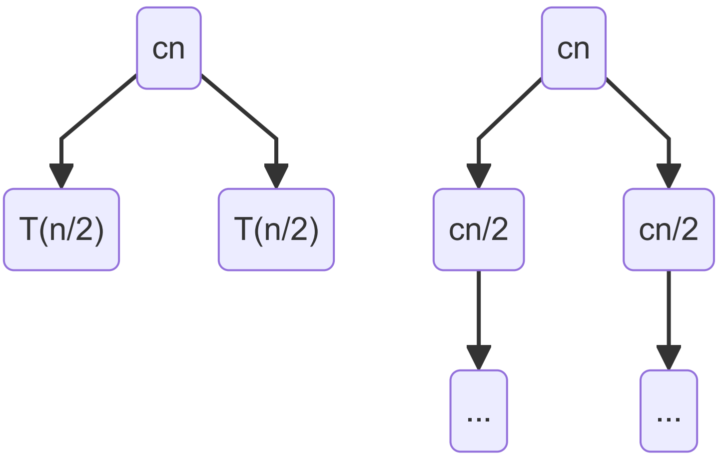


The recursion tree has a depth of n giving us the sum of $cn + c(n-1) + \ldots + c$ or $cn\frac{1(n+1)}{2} = \Theta(n^2)$.

Best Case

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + \Theta(n) \\
&= 2T(\frac{n}{2}) + cn
\end{aligned}
$$

The best case is that our partition has an equal number of elements on both sides of the array.



The recursion tree has a depth of $\lg n$ with each layer having a cost of cn giving us a total of $cn \lg n$ or $\Theta(n \lg n)$

Average Case

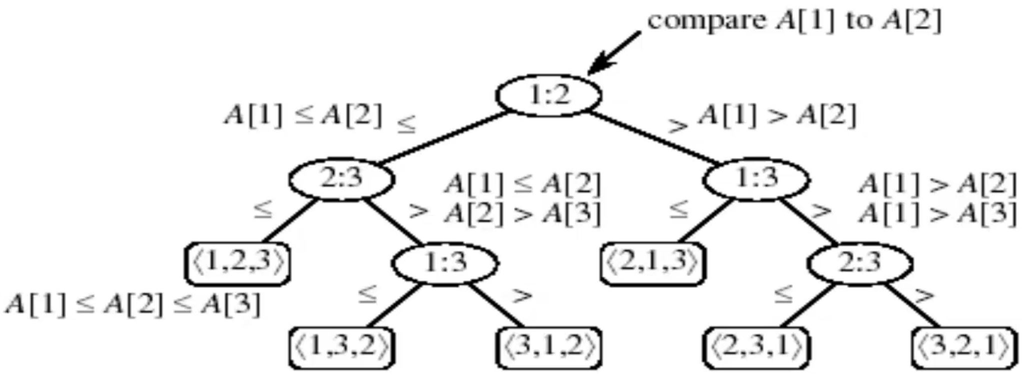
- the average case will also be $\Theta(n \lg n)$
- notice that regardless of the split such as $\frac{1}{3}\frac{2}{3}$ split or $\frac{1}{10}\frac{9}{10}$ split, the asymptotic bound will also be $O(n \lg n)$

Lower Bound on Sorting

Claim: when using **comparison sorts** (i.e. insertion sort, merge sort, heapsort, quicksort, etc...) we must make *at least* $n \lg n$ comparisons in the general case.

The Decision Tree Model

- an abstraction of any comparison sort
- represents comparisons made by a specific algorithm on inputs of a given size



- each internal node is labeled by indices of array elements from their original position
- each leaf is labeled by the permutation of orders that the algorithm determines

Lower bound of the tree

The lower bound of the height of the tree is $n \lg n$.

Counting Sort

Counting sort is a linear time sorting algorithm.

Code

```
function countingSort(A: number[], max: number) {
    const C = new Array(max), B = new Array(A.length);
    for (let i = 0; i < C.length; i++) C[i] = 0;
    // count occurrences of each value
    for (let i = 0; i < A.length; i++) C[A[i]]++;
    // accumulate
    for (let i = 1; i < max; i++) C[i] = C[i] + C[i - 1];
    for (let i = A.length - 1; i > 0; i--) {
        B[C[A[i]]] = A[i];
        C[A[i]]--;
    }
    return B;
}
```

Design

- counting sort assumes that each of the n input elements is an integer in the range $[0, k]$ for some integer k
    - when $k = O(n)$, the sort runs in $T(n)$
    - best if $K << n$
- for each input element x, count how many elements are less than x
    - this information can be used to place x directly into its position in the output array
- does not sort in place
    - needs a 2$^{\text{nd}}$ array of size k and and 3$^{\text{rd}}$ array of size n
- it is stable

Stable Algorithms

Numbers with the *same* value appear in the output array in the same order as they do in the input array.

Stability of some other sorting algorithms

- insertion sort ✅
- quicksort ❌

Runtime Analysis

$$\Theta(n + k)$$

Where k is `max` and when $k = O(n)$ in most cases then the entire complexity is $\Theta(n)$.

Radix Sort

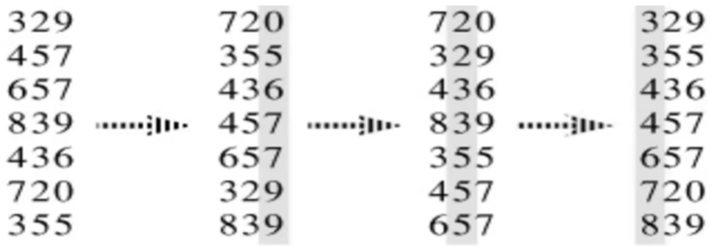
Radix sort is a linear time sorting algorithm.

Code

```
// TODO
```

Design

- the procedure assumes that each element in the n-element array `A` has d digits $D_1 D_2 ... D_d$
  - where digit D_1 is the lowest order digit
  - and digit D_d is the highest order digit
- we use counting sort (or any other stable sort) on each digit
  - the type of sort you use here will effect its complexity
- sort from D_1 to D_d



Runtime Analysis

Given n d-digit numbers in which each digit can take up to k possible values, radix sort sorts in $\Theta(d * (k + n))$ if the stable sort used uses $\Theta(n + k)$.

Bucket Sort

Code

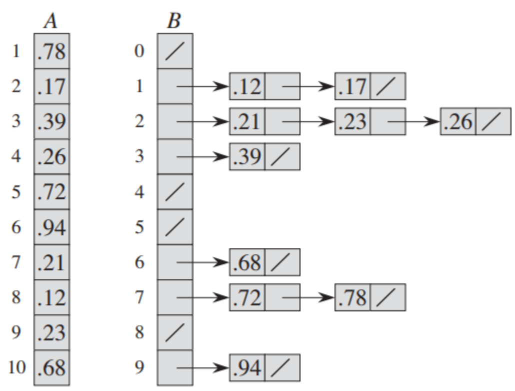
```
BUCKET-SORT(A)
let B[0 .. n − 1] be a new array
n = A.length
for i = 0 to n − 1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n − 1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

Design

- bucket sort assumes the input has elements evenly distributed over the interval 0 and 1
- divide the interval into n equal sized sub-interval buckets in an array B
    - where each element in B is the head of a linked list (i.e. a bucket)
- distribute the input elements into the buckets
    - for a bucket i, it covers the domain of $[i \times \frac{1}{n}, (i+1)\frac{1}{n}]$
    - if an element has a value a, its bucket index is

$$i \times \frac{1}{n} \leq a \leq (i+1)\frac{1}{n}$$
$$i \leq a * n \leq i + 1$$
$$i = \lfloor a \times n \rfloor$$

- sort each bucket with insertion sort
- go through each bucket to list the elements as a sorted array



(a) (b)

Runtime Analysis

Worst Case

In the worst case, all the elements are placed in the same bucket and the runtime is $O(n^2)$

Average Case

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Where n_i is the number of elements that fall into bucket i. We will use expectation for the average case.

$$\begin{aligned} E[T(n)] &= E[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E(n_i^2)) \end{aligned}$$

Use X_{ij} as a RV that `A[j]` falls into bucket i.

$$X_{ij} = \begin{cases} 0 & \text{with probability } 1 - \frac{1}{n} \\ 1 & \text{with probability } \frac{1}{n} \end{cases}$$

49:33 L11

Order Statistics

i^{th} Smallest Element of the Array

Code

```
RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]
q = RANDOMIZED-PARTITION(A, p, r)
k = q - p + 1
if i == k                // the pivot value is the answer
    return A[q]
elseif i < k
    return RANDOMIZED-SELECT(A, p, q - 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

Runtime Analysis

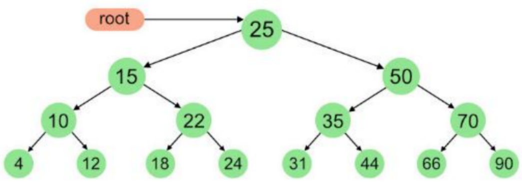
$$E(T(n)) = O(n)$$

beginning of L12

Binary Search Tree (BST)

- each node contains a quintuple
  - an index
  - a key
  - pointers to its left, right child, and parent
- all keys in the left subtree of x should be less than or equal to that of x
  - and all in right subtree should greater than or equal to that of x
- search, insert, delete, predecessor, successor, minimum, maximum operations are all $O(h)$ where h is the height of the tree
- in a standard BST, h is determined by the order of inserting n items
  - the best case $h = n \lg n$
  - the worst case $h = n$

Tree Traversals



In Order

`4, 10, 12, 15, 18, 22, 24, ...`

1. left subtree
2. root
3. right subtree

Pre-Order

`25, 15, 10, 4, 12, 22, 50, 35, 31, 44, 70, 66, 90`

1. root
2. left subtree
3. right subtree

Post-Order

`4, 12, 10, 18, 24, 22, 15, 32, 44, 35, 66, 90, 70, 50, 25`

1. left subtree
2. right subtree
3. root

Searching

```
TREE－MAX(x)
        While  x. right≠NIL
                x=x. right
        returnx
```

```
TREE－MIN(x)
        While  x. left≠NIL
                x=x. left
        returnx
```

Successor

```
TREE-SUCCESSOR(x)
if right[x] ≠ NIL
   then return TREE-MINIMUM (right[x])

y = parent[x]

while y ≠ NIL and x = right[y]
   x = y
   y = parent[y]

return y
```

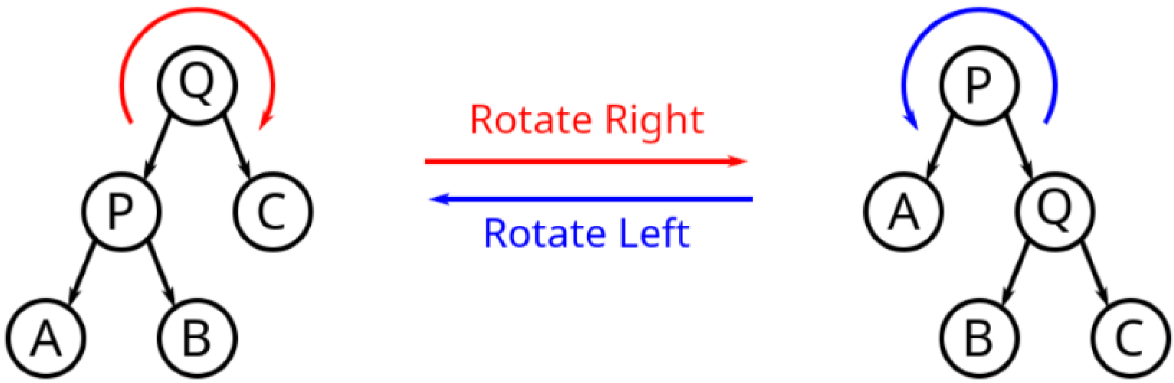
Insert

```
TREE-INSERT(T, z)
        y=NIL
        x=T.root
        while x≠NIL
                y = x
                if z.key<x.key
                        x=x.left
                else   x=x.right
        z.p=y
        if y==NIL
                T.root=z
        elseif z.key <y.key
                y.left=z
        else   y.right=z
```

Delete

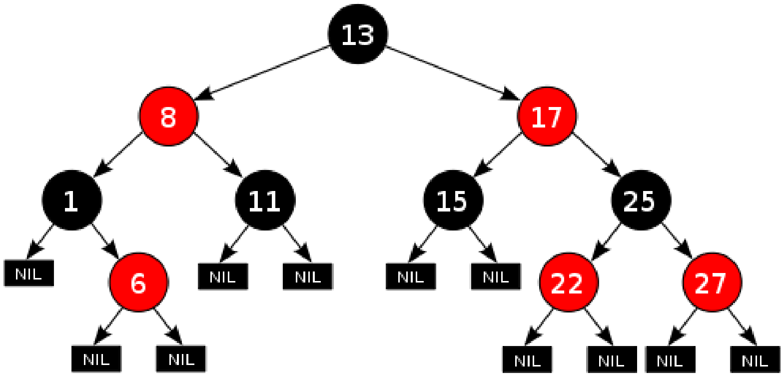
1. `z` has not children
    - just remove `z`
2. `z` has 1 child
    - replace `z` with its child
3. `z` has 2 children
    - replace `z` with its successor

Rotation




```
# Right rotation pseudocode
function rightRotate(y):
    x = y.left
    T = x.right
    # Perform rotation
    x.right = y
    y.left = T
    return x

# Left rotation pseudocode
function leftRotate(x):
    y = x.right
    T = y.left
    # Perform rotation
    y.left = x
    x.right = T
    return y
```



Source: wikipedia.org

Red-Black Tree Properties (Definition of RB Trees)

A red-black tree is a BST with following properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf is NIL and black.
4. Both children of each red node are black.
5. All root-to-leaf paths contain the same number of black nodes.

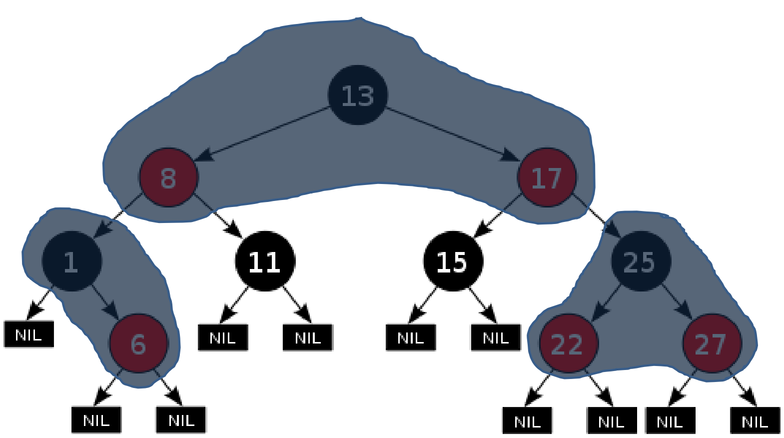
More Properties

1. No root-to-leaf path contains two consecutive red nodes.
2. For each node *x*, all paths from *x* to descendant leaves contain the same number of black nodes. This number, not counting *x*, is the black height of *x*, denoted *bh(x)*.
3. No root-to-leaf path is more than twice as long as any other.

Theorem. A red-black tree with *n* internal nodes has height $\leq 2\log(n+1)$.

Proof of Theorem

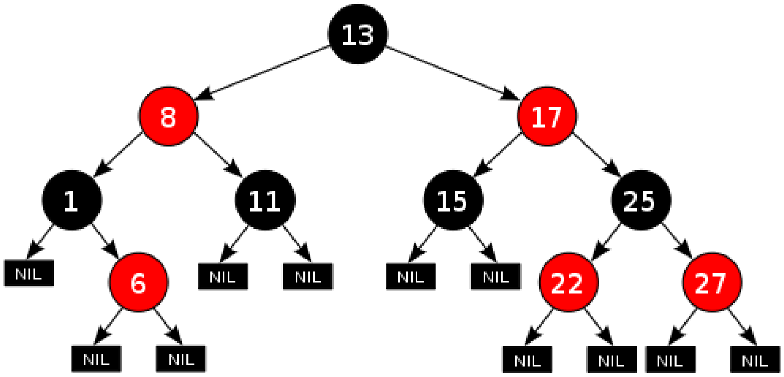
- Consider a red black tree with height h.
- Collapse all red nodes into their (black) parent nodes to get a tree with all black nodes.
- Each internal node has 2 to 4 children.
- The height of the collapsed tree is $h' \geq h/2$, and all external nodes are at the same level.
- Number of internal nodes in collapsed tree is
  $$n \geq 1 + 2 + 2^2 + \cdots + 2^{h'-1} = 2^{h'} - 1 \geq 2^{h/2} - 1.$$
- So, $h \leq 2\log_2(n+1)$.



Source: wikipedia.org

Insert a node z

- Insert *z* as in a regular BST; color it red.
- If any violation to RB properties, fix it.
- Possible violations:
  - The root is red. (Case 0)
    To fix up, make it black.
  - Both z and z's parent are red.
    To fix up, consider three cases. (Actually, six cases: I, II, III, I', II', III')

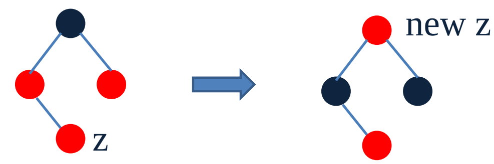


Source: wikipedia.org

Insert Fixup: Case I

The parent and "uncle" of z are both red:

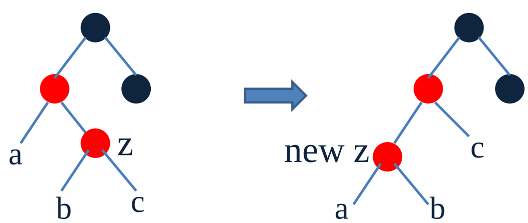
- Color the parent and uncle of z black;
- Color the grandparent of z red;
- Repeat on the grandparent of z.



Insert Fixup: Case II

The parent of z is red, the uncle of z is black,
z's parent is a left child , z is a right child:

- Left Rotate on z's parent;
- Make z's left child the new z; it becomes Case III.

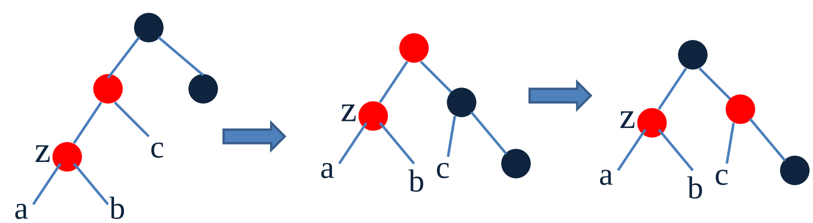


Case III

Insert Fixup: Case III

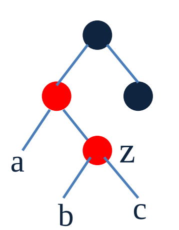
The parent of z is Red and the "uncle" is Black,
z is a left child, and its parent is a left child:

- Right Rotate on the grandparent of z.
- Switch colors of z's parent and z's sibling.
- Done!

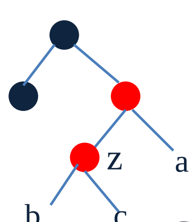


Case II'

Symmetric to Case II



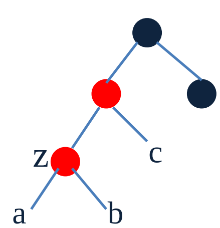
Case II



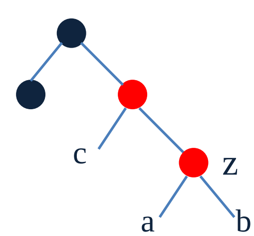
Case II'

Case III'

Symmetric to Case III



Case III



Case III'

Demonstration

- http://www.ece.uc.edu/~franco/C321/html/RedBlack/redblack.html

BST Deletion Revisited

Delete z

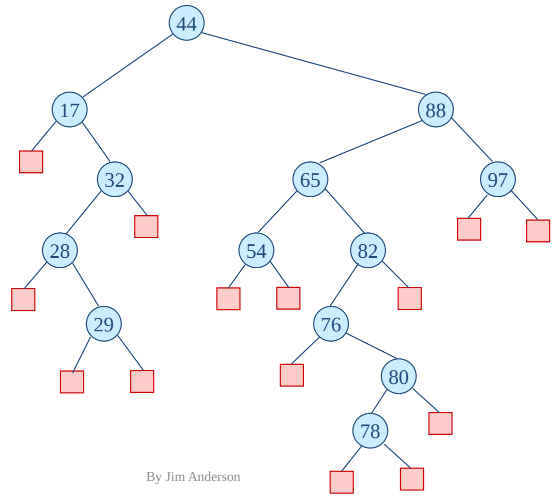
- If z has no children, we just remove it.
- If z has only one child, we splice out z.
- If z has two children, we splice out its successor y, and then replace z's key and satellite data with y's key and satellite data.

- Which physical node is deleted from the tree?

BST: Delete

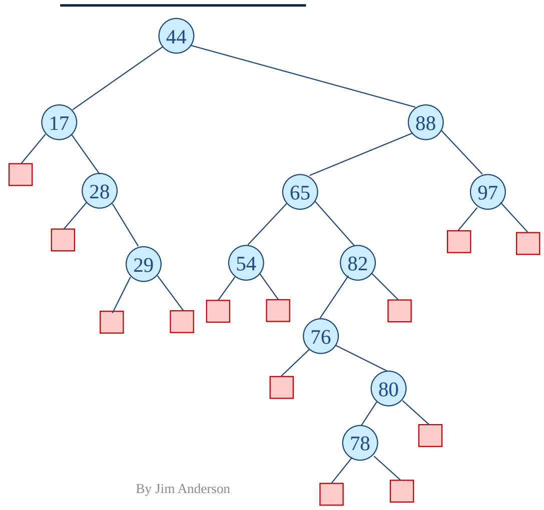
Delete(32)

Has only one child: just splice out 32.



BST: Delete

Delete(32)

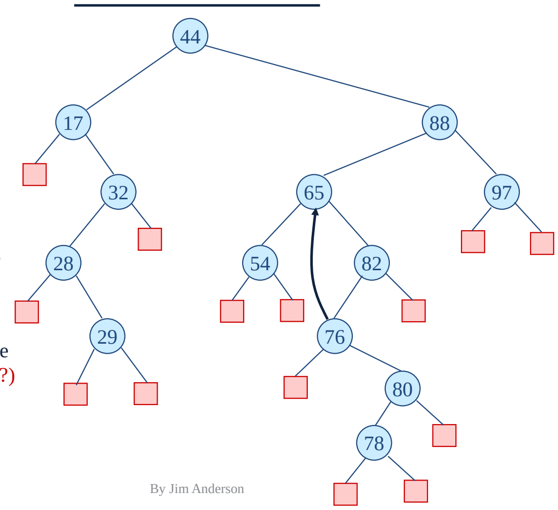


BST: Delete

Delete(65)

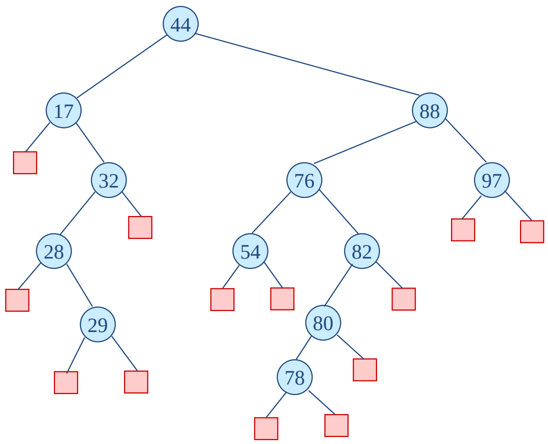
Has two children: Replace 65 by successor, 76, and splice out successor.

Note: Successor can have at most one child. (Why?)

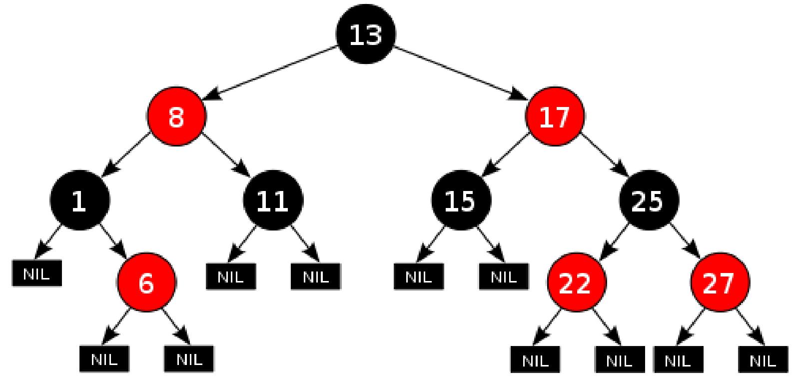


BST: Delete

Delete(65)



By Jim Anderson



Source: wikipedia.org

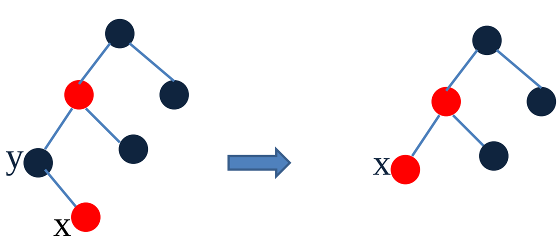
Delete *z*

- Delete z as in a regular BST.
- If z had two (non-nil) children, when copying y's key and satellite data to z, do not copy the color, (i.e., keep z's color).
- Let y be the node being removed or spliced out.
  (Note: either y = z or y = successor(z) .)
- If y is red, no violation to the red-black properties.
- If y is black, then one or more violations may arise and we need to restore the red-black properties.

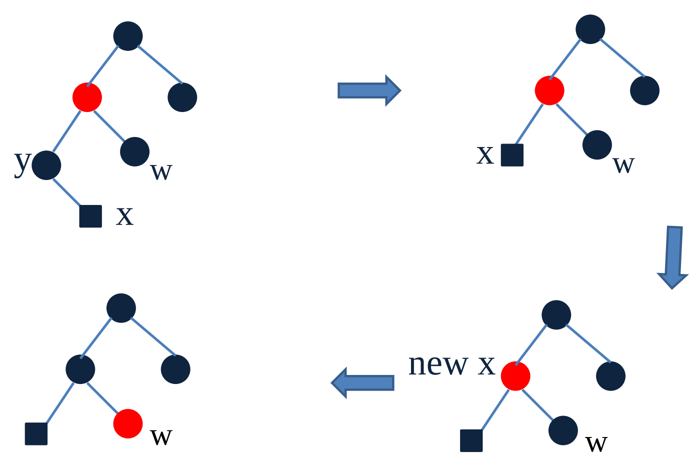
- Let x denote the child of y before it was spliced out.
- x is either nil (leaf) or was the only non-nil child of y.

Restoring RB Properties

- Easy: If x is red, just change it to black.
- More involved: If x is black.



Example: x is black



Restoring RB Properties

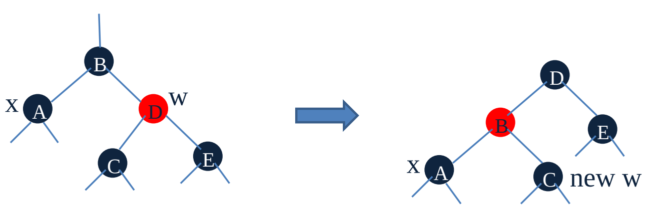
- Assume: x is black and is a left child.
- The case where x is black and a right child is similar (symmetric).
- Four cases:
  1. x' sibling w is red.
  2. x's sibling w is black; both children of w are black.
  3. x's sibling w is black; left child of w is red, right child black.
  4. x's sibling w is black; right child of w is red.

Main idea

- Regard the pointer x itself as black.
- Counting x, the tree satisfies RB properties.
- Transform the tree and move x up until:
  - x points to a red node, or
  - x is the root, or
  - RB properties are restored.
- At any time, maintain RB properties, with x counted as black.

x is a black left child: Case 1

- x' sibling w is red.
- Left rotate on B; change colors of B and D.
- Transform to Case 2, 3, or 4 (where w is black).

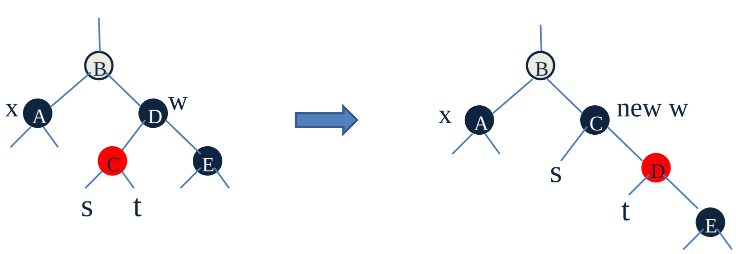


x is a black left child : Case 2

- x's sibling w is black; both children of w are black.
- Move x up, and change w's color to red.
- If new x is red, change it to black; else, repeat.

x is a black left child : Case 3

- x's sibling w is black; w's left child is red, right child black.
- Right rotate on w (D); switch colors of C and D.
- C becomes the new w.
- Transform to Case 4.



X is a black left child : Case 4

- x's sibling w is black; w's right child is red.
- Left rotate on B; switch colors of B and D; change E to black.
- Done!