FR-0: **Big-$O$ Notation**

$$\begin{aligned} O(g(n)) &= \{f(n) \mid \exists c, n_0, s.t. \\ &\quad f(n) \leq cg(n) \quad \text{whenever} \quad n > n_0\} \end{aligned}$$

$f(n) \in O(g(n))$ means:

- $f$ is bound from above by $g$
- $f$ grows no faster than $g$
- $g$ is an upper bound on $f$

FR-1: **Big-$\Omega$ Notation**

$$\begin{aligned} \Omega(g(n)) &= \{f(n) \mid \exists c, n_0, s.t. \\ &\quad cf(n) \geq g(n) \quad \text{whenever} \quad n > n_0\} \end{aligned}$$

$f(n) \in \Omega(g(n))$ means:

- $f$ is bound from below by $g$
- $g$ grows no faster than $f$
- $g$ is a lower bound on $f$

FR-2: **Big-$\Theta$ Notation**

$$\begin{aligned} \Theta(g(n)) &= \{f(n) \mid \exists c_1, c_2, n_0, s.t. \\ &\quad c_1 g(n) \leq f(n) \leq c_2 g(n) \\ &\quad \text{whenever} \quad n > n_0\} \end{aligned}$$

Alternately,

$$\Theta(g(n)) = O(g(n)) \bigcap \Omega(g(n))$$

FR-3: **Big-$\Theta$ Notation**

Show:

$$3n^2 + 4n \in \Theta(n^2)$$

FR-4: **Big-$\Theta$ Notation**

Show:

$$3n^2 + 4n \in \Theta(n^2)$$

$c_1 * n^2 \leq 3n^2 + 4n \leq c_2 * n^2$

True, as long as $c_1 \leq 3 + 4/n$, $c + 2 \geq 3 + 4/n$

(since $n > n_0$, we can assume $4/n \leq 1$)

FR-5: **little-$o$**

$$\begin{aligned} o(g(n)) &= \{f(n) \mid \forall c > 0, \exists n_0 > 0 \ s.t. \\ &\quad 0 \leq f(n) < cg(n) \text{ whenever } n \geq n_0\} \end{aligned}$$

$$f(n) \in o(g(n)) \Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

FR-6: **little-$\omega$**

$$\begin{aligned} \omega(g(n)) &= \{f(n) \mid \forall c > 0, \exists n_0 > 0 \ s.t. \\ &\quad 0 \leq cg(n) < f(n) \text{ whenever } n \geq n_0\} \end{aligned}$$

$$f(n) \in \omega(g(n)) \Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

FR-7: **Summations**

- We will assume you've seen inductive proof that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ (check Appendix 1 in the text otherwise!)
- Can use induction to prove bounds as well. Show:

$$\sum_{i=1}^{n} 3^i + 2^i = O(3^n)$$

FR-8: **Summations**

$\sum_{i=1}^{n} 3^i + 2^i = O(3^n)$

- Base Case:
  - $\sum_{i=1}^{1} 3^i + 2^i = 3 + 2 \leq c * (3^1)$ as long as $c \geq 5/3$

FR-9: **Summations** $\sum_{i=1}^{n} 3^i + 2^i = O(3^n)$

- Recursive Case:

$$\begin{aligned} \sum_{i=1}^{n+1} 3^i + 2^i &= (\sum_{i=1}^{n}(3^i + 2^i) + 3^{n+1} + 2^{n+1} \\ &\leq c3^n + 3^{n+1} + 2^{n+1} \\ &\leq c3^n + 3^{n+1} + 3^{n+1} \\ &= c3^n + 2 * 3^{n+1} \\ &= c/3 * 3^{n+1} + 2 * 3^{n+1} \\ &= (1/3 + 2/c)c3^{n+1} \\ &\leq c3^{n+1} \end{aligned}$$

As long as $(1/3 + 2/c) \leq 1$, or $c \geq 3$

FR-10: **Bounding Summations**

$$\sum_{i=1}^{n} a_i \leq n * a_{max}$$

for instance: $\sum_{i=1}^{n} i \leq n^2 \in O(n^2)$

and

$$\sum_{i=1}^{n} a_i \geq n * a_{min}$$

for instance: $\sum_{i=1}^{n} i \geq 1 * n \in \Omega(n)$

(note that the bounds are not always tight!)

FR-11: **Splitting Summations**

We can sometimes get tighter bounds by splitting the summation:

$$\begin{aligned}
\sum_{i=1}^{n} \lg i &= \sum_{i=1}^{\lfloor n/2 \rfloor} \lg i + \sum_{i=\lfloor n/2 \rfloor + 1}^{n} \lg i \\
&\geq n/2 * \lg 1 + n/2 * \lg n/2 \\
&= n/2(\lg n - \lg 2) \\
&= n/2(\lg n - 1) \\
&\in \Omega(n \lg n)
\end{aligned}$$

FR-12: **Splitting Summations**

We can split summations in more tricky ways, as well. Consider the harmonic series:

$$H_n = \sum_{i=1}^{n} \frac{1}{n}$$

How could we split this to get a good upper bound?

*HINT:* Break sequence into pieces that are all bounded by the same amount – if the series gets smaller, size of each piece should increase.

FR-13: **Splitting Summations**

$$\begin{aligned} H_n &= \sum_{i=1}^{n} \frac{1}{i} \\ &\leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i + j} \\ &\leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i} \\ &\leq \sum_{i=0}^{\lfloor \lg n \rfloor} 1 \\ &\leq \lg n + 1 \end{aligned}$$

FR-14: **Recurrence Relations**

```
MergeSort(A,low,high) {
   if (low < high + 1) {
      mid1 = floor ( (low + high) / 3) )
      mid2 = floor ( 2(low + high) / 3) )
      MergeSort(A,low,mid1)
      MergeSort(A,mid+1,mid2)
      MergeSort(A,mid2+1,high)
      Merge3(A,low,mid1,mid2,high)
   }
}
```

FR-15: **Recurrence Relations**

```
MergeSort(A,low,high) {
   if (low < high + 1) {
      mid1 = floor ( (low + high) / 3) )
      mid2 = floor ( 2(low + high) / 3) )
      MergeSort(A,low,mid1)
      MergeSort(A,mid+1,mid2)
      MergeSort(A,mid2+1,high)
      Merge3(A,low,mid1,mid2,high)
   }
}
```

$$\begin{aligned} T(0) &= \Theta(1) \\ T(1) &= \Theta(1) \\ T(2) &= \Theta(1) \\ T(n) &= T\left(\left\lceil \frac{n}{3} \right\rceil\right) + 2 * T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + \Theta(n) \end{aligned}$$

FR-16: **Recurrence Relations**

- How do we solve recurrence relations?
  - Substitution Method
    - Guess a solution

- Prove the guess is correct, using induction

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 3T\left(\frac{n}{3}\right) + n \end{aligned}$$

FR-17: **Substitution Method**

- Inductive Case

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{3}\right) + n \\ &\leq 3\left(c\frac{n}{3}\lg\frac{n}{3}\right) + n \\ &= cn\lg n - cn\lg 3 + n \\ &\leq cn\lg n \end{aligned}$$

FR-18: **Substitution Method**

- Base Case

$$\begin{aligned} T(1) &= 1 \\ T(n) &\leq cn\lg n \\ T(1) &\leq c * 1 * \lg 1 \\ T(1) &\leq c * 1 * 0 = 0 \end{aligned}$$

Whoops! If the base case doesn't work the inductive proof is broken! What can we do?
FR-19: **Substitution Method**

- Fixing the base case

Note that we only care about $n > n_0$, and for $n \geq 6$, recurrence does not depend upon $T(1)$ except through $T(2) - -T(5)$ and $T(3)$

$$\begin{aligned} T(2) &\leq 2 * c * \lg 2 \\ T(3) &\leq 3 * c * \lg 3 \\ T(4) &\leq 4 * c * \lg 2 \\ T(5) &\leq 5 * c * \lg 3 \end{aligned}$$

(for sufficiently large $c$) FR-20: **Substitution Method**

- Sometimes, the math doesn't work out in the substitution method:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2 * T\left(\frac{n}{2}\right) + 1 \end{aligned}$$

FR-21: **Substitution Method** Try $T(n) \leq cn$:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 1 \\ &\leq c2\left(\frac{n}{2}\right) + 1 \\ &\leq cn + 1 \end{aligned}$$

We did not get back $T(n) \leq cn$ – that extra +1 term means the proof is not valid. We need to get back *exactly* what we started with (see invalid proof of $\sum_{i=1}^{n} i \in O(n)$ for why this is true)
FR-22: **Substitution Method** Try $T(n) \leq cn - b$:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 1 \\ &\leq 2 * \left(c\frac{n}{2} - b\right) + 1 \\ &\leq cn - 2b + 1 \\ &\leq cn - b \end{aligned}$$

As long as $b \geq 1$
FR-23: **Recursion Trees**

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

FR-24: **Recursion Trees**

$$T(n) = T(n-1) + cn$$

FR-25: **Recursion Trees**

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

FR-26: **Recursion Trees**

$$T(n) = 3T(n/4) + cn^2$$

FR-27: **Recursion Trees**

$$T(n) = \qquad cn^2 \qquad +$$

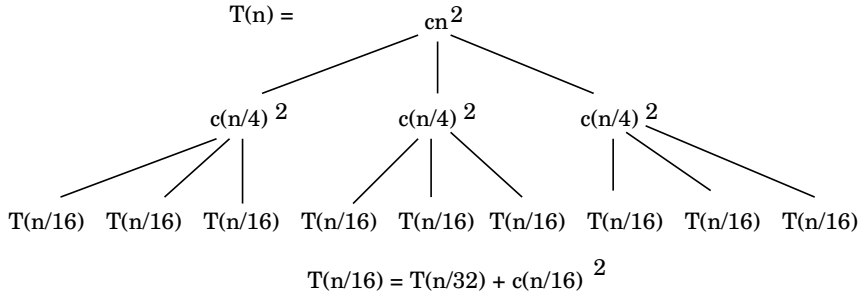
$$T(n/4) \quad + \quad T(n/4) \quad + \quad T(n/4)$$

FR-28: **Recursion Trees**

$$T(n) = cn^2 + T(n/4) + T(n/4) + T(n/4)$$

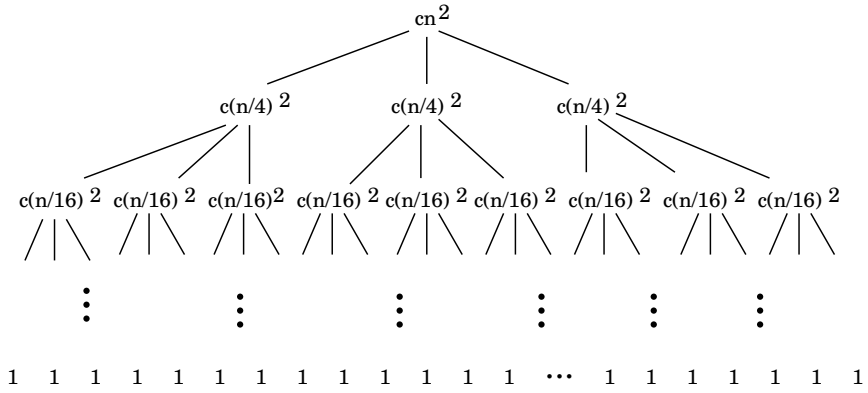
$$T(n/4) = 3T(n/16) + c(n/4)^2$$

FR-29: **Recursion Trees**

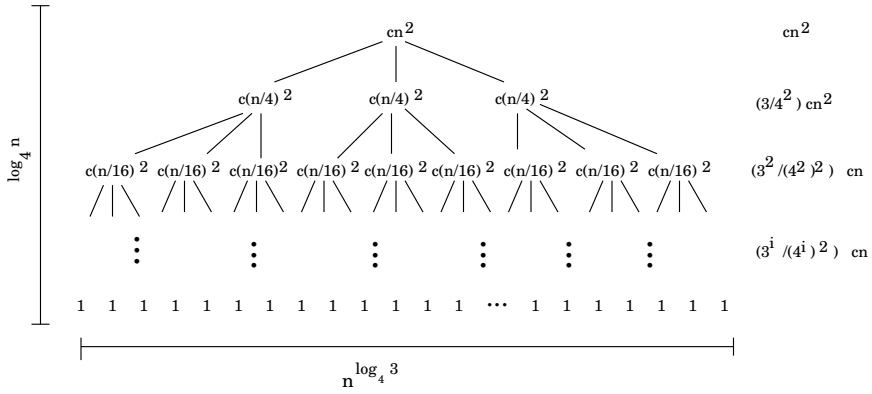



$$T(n/16) = T(n/32) + c(n/16)^2$$

FR-30: **Recursion Trees**




FR-31: **Recursion Trees**



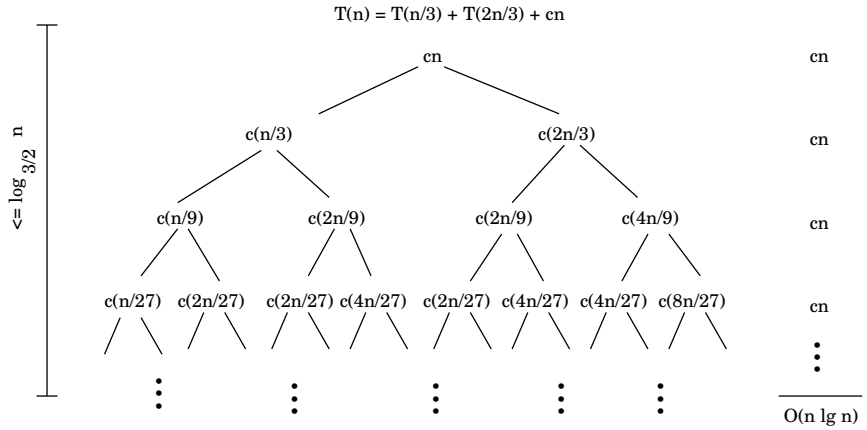

FR-32: **Recursion Trees**

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n} \left(\frac{3^i}{4^{2i}}\right)^i cn^2 + \sum_{i=0}^{n^{\log_4 3}} 1 \\
&< \sum_{i=0}^{\log_4 n} \left(\frac{3}{4}\right)^i cn^2 + n^{\log_4 3} \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i cn^2 + n^{\log_4 3} \\
&= \frac{1}{1-3/4} cn^2 + n^{\log_4 3} \\
&= 4cn^2 + n^{\log_4 3} \\
&\in O(n^2)
\end{aligned}
$$

(now prove bound using substitution method) FR-33: **Recursion Trees**

T(n) = T(n/3) + T(2n/3) + cn

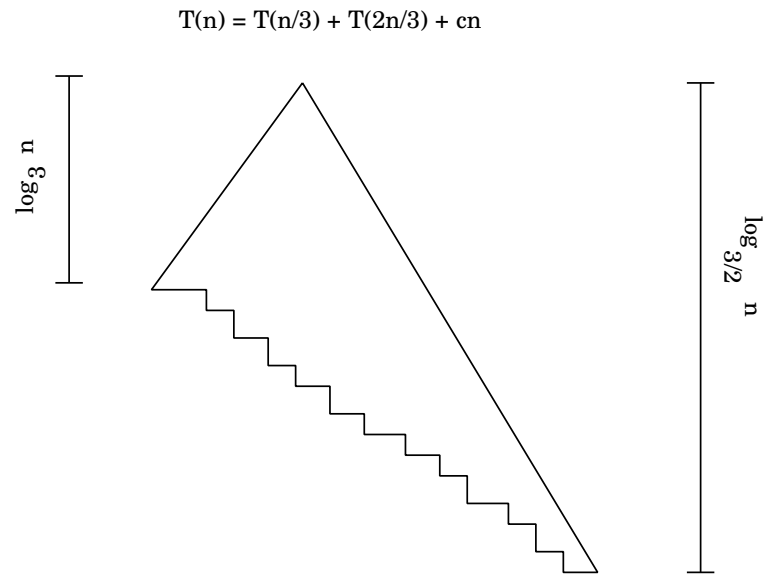
FR-34: **Recursion Trees**



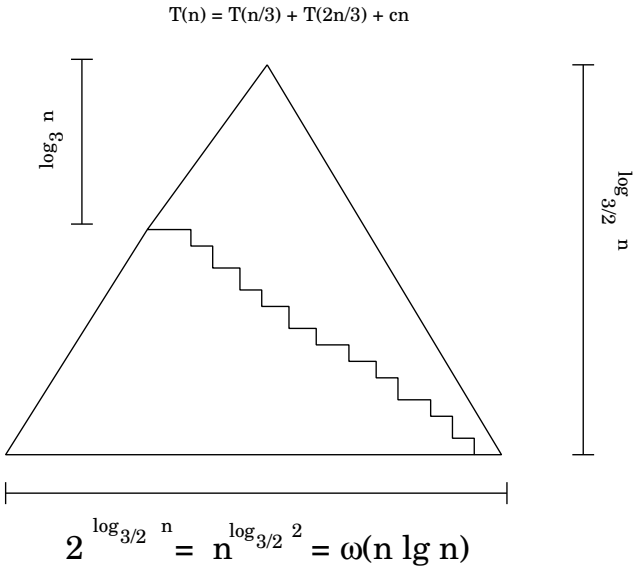

FR-35: **Recursion Trees**

- There is a small problem – this tree is actually irregular in shape!

FR-36: **Recursion Trees**




FR-37: **Recursion Trees**




FR-38: **Recursion Trees**

- If we are only using recursion trees to create a guess (that we will later verify using substitution method), then we can be a little sloppy.
- Show $T(n) = T(n/3) + T(2n/3) + cn \in O(n \lg n)$

FR-39: **Recursion Trees**

- $T(n) = T(n-2) + n^2$
- $T(n) = 4T(n/4) + n \lg n$

FR-40: **Renaming Variables**

- Consider:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T\left(\sqrt{n}\right) + \lg n \end{aligned}$$

- The $\sqrt{}$ is pretty ugly – how can we make it go away?
- Rename variables!

FR-41: **Renaming Variables**

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T\left(\sqrt{n}\right) + \lg n \end{aligned}$$

Let $m = \lg n$, (and so $n = 2^m$)

$$\begin{aligned} T(2^m) &= 2T\left(\sqrt{2^m}\right) + \lg 2^m \\ &= 2T\left(2^{m/2}\right) + m \end{aligned}$$

FR-42: **Renaming Variables**

$$T(2^m) = 2T\left(2^{m/2}\right) + m$$

Now let $S(m) = T(2^m)$

$$\begin{aligned} S(m) &= T(2^m) \\ &= 2T\left(2^{m/2}\right) + m \\ &= 2S\left(\frac{m}{2}\right) + m \end{aligned}$$

FR-43: **Renaming Variables**

$$\begin{aligned} S(m) &= 2S\left(\frac{m}{2}\right) + m \\ &\leq cm \lg m \end{aligned}$$

So:

$$\begin{aligned} T(n) &= T(2^m) \\ &= S(m) \\ &\leq cm \lg m \\ &= c \lg n \lg \lg n \end{aligned}$$

FR-44: **Master Method**

$$T(n) = aT(n/b) + f(n)$$

1. if $f(n) \in O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$
2. if $f(n) \in \Theta(n^{\log_b a})$ then $T(n) \in \Theta(n^{\log_b a} * \lg n)$
3. if $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c < 1$ and large $n$, then $T(n) \in \Theta(f(n))$

FR-45: **Master Method**

$$T(n) = 9T(n/3) + n$$

FR-46: **Master Method**

$$T(n) = 9T(n/3) + n$$

- $a = 9, b = 3, f(n) = n$
- $n^{\log_b a} = n^{\log_3 9} = n^2$
- $n \in O(n^{2-\epsilon})$

$T(n) = \Theta(n^2)$

FR-47: **Master Method**

$$T(n) = T(2n/3) + 1$$

FR-48: **Master Method**

$$T(n) = T(2n/3) + 1$$

- $a = 1, b = 3/2, f(n) = 1$
- $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$
- $1 \in O(1)$

$T(n) = \Theta(1 * \lg n) = \Theta(\lg n)$

FR-49: **Master Method**

$$T(n) = 3T(n/4) + n \lg n$$

FR-50: **Master Method**

$$T(n) = 3T(n/4) + n \lg n$$

- $a = 3, b = 4, f(n) = n \lg n$
- $n^{\log_b a} = n^{\log_4 3} = n^{0.792}$
- $n \lg n \in \Omega(n^{0.792+\epsilon})$
- $3(n/4) \lg(n/4) \leq c * n \lg n$

$T(n) \in \Theta(n \lg n)$

FR-51: **Master Method**

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

FR-52: **Master Method**

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

- $a = 2, b = 2, f(n) = n \lg n$
- $n^{\log_b a} = n^{\log_2 2} = n^1$

Master method does not apply!
$n^{1+\epsilon}$ grows faster than $n \lg n$ for *any* $\epsilon > 0$
Logs grow *incredibly* slowly! $\lg n \in o(n^\epsilon)$ for any $\epsilon > 0$

FR-53: **Probability Review**

- Indicator variable associated with event $A$:

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs} \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

- Example: Flip a coin: $Y$ is a random variable representing the coin flip

$$X_H = I\{Y = H\} = \begin{cases} 1 & \text{if } Y = H \\ 0 & \text{otherwise} \end{cases}$$

FR-54: **Probability Review**

- Expected value $E[]$ of a random variable
  - Value you "expect" a random variable to have
  - Average (mean) value of the variable over many trials
  - Does not have to equal the value of any particular trial
    - Bus example(s)

FR-55: **Probability Review**

- Expected value $E[]$ of a random variable

$$E[X] = \sum_{\text{all values } x \text{ of } X} x * Pr\{X = x\}$$

- When we want the "average case" running time of an algorithm, we want the Expected Value of the running time

FR-56: **Probability Review**

$X_H = I\{Y = H\}$

$$\begin{aligned} E[X_H] &= E[I\{Y = H\}] \\ &= 1 * Pr\{Y = H\} + 0 * Pr\{Y = T\} \\ &= 1 * 1/2 + 0 * 1/2 \\ &= 1/2 \end{aligned}$$

FR-57: **Probability Review**

- Expected # of heads in $n$ coin flips
  - $X$ = # of heads in $n$ flips
  - $X_i$ = indicator variable: coin flip $i$ is heads

FR-58: **Probability Review**

- Expected # of heads in $n$ coin flips
  - $X$ = # of heads in $n$ flips
  - $X_i$ = indicator variable: coin flip $i$ is heads

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[X_i] \\ &= \sum_{i=1}^{n} \frac{1}{2} = \frac{n}{2} \end{aligned}$$

FR-59: **Probability Review**

- For any event $A$, indicator variable $X_A = I\{A\}$ $E[X_A] = Pr\{A\}$

$$\begin{aligned} E[X_A] &= 1 * Pr\{A\} + 0 * Pr\{\neg A\} \\ &= Pr\{A\} \end{aligned}$$

FR-60: **Hiring Problem**

- Calculate the expected number of hirings
  - $X$ = # of candidates hired
  - $X_i = I\{\text{Candidate } i \text{ is hired}\}$
  - $X = X_1 + X_2 + \ldots + X_n$

$$E[X] =$$

FR-61: **Hiring Problem**

- Calculate the expected number of hirings
  - $X$ = # of candidates hired
  - $X_i = I\{\text{Candidate } i \text{ is hired}\}$
  - $X = X_1 + X_2 + \ldots + X_n$

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[x_i] \end{aligned}$$

- What is $E[X_i]$?

FR-62: **Hiring Problem**

- What is $E[X_i]$?
  - $E[X_i]$ = Probability that the $i$th candidate is hired
  - When is the $i$th candidate hired?

FR-63: **Hiring Problem**

- What is $E[X_i]$?
  - $E[X_i]$ = Probability that the $i$th candidate is hired
  - $i$th candidate hired when s/he is better than the $i-1$ candidates that came before
  - Assuming that all permutations of candidates are equally likely, what is the probability that the $i$th candidate is the best of the first $i$ candidates?

FR-64: **Hiring Problem**

- What is $E[X_i]$?
  - $E[X_i]$ = Probability that the $i$th candidate is hired
  - $i$th candidate hired when s/he is better than the $i-1$ candidates that came before
  - Assuming that all permutations of candidates are equally likely, what is the probability that the $i$th candidate is the best of the first $i$ candidates?
    - $\frac{1}{i}$

FR-65: **Hiring Problem**
Probability that the $i$th candidate is best of first $i$ is $\frac{1}{i}$

- Sanity Check: (Doing a few concrete examples as a sanity check is often a good idea)
  - $i = 1$, probability that the first candidate is the best so far = 1/1 = 1
  - $i = 2$: (1,2), (2,1) In one of the two permutations, 2nd candidate is the best so far

- $i = 3$: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1) In two of the 6 permutations, the 3rd candidate is the best so far

- Note that a few concrete examples do not *prove* anything, but a counter-example can show that you have made a mistake

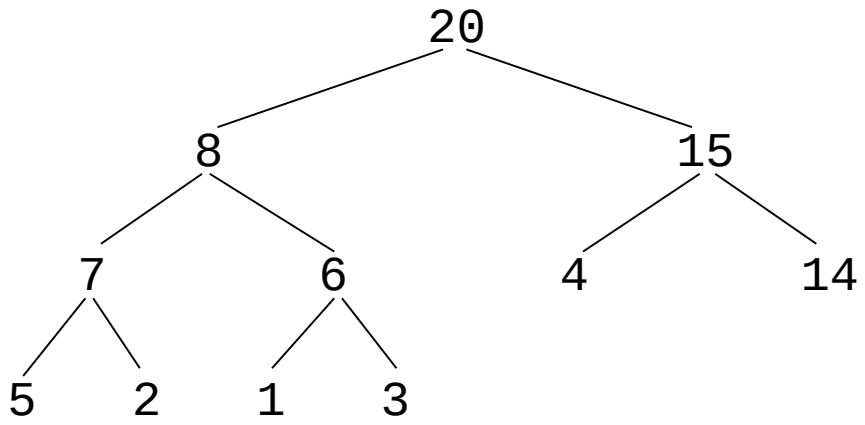
FR-66: **Hiring Problem**

- Now that we know that $E[X_i] = \frac{1}{i}$, we can find the expected number of hires:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[x_i] \\ &= \sum_{i=1}^{n} 1/i \\ &= \ln n + O(1) \\ &\in O(\lg n) \end{aligned}$$

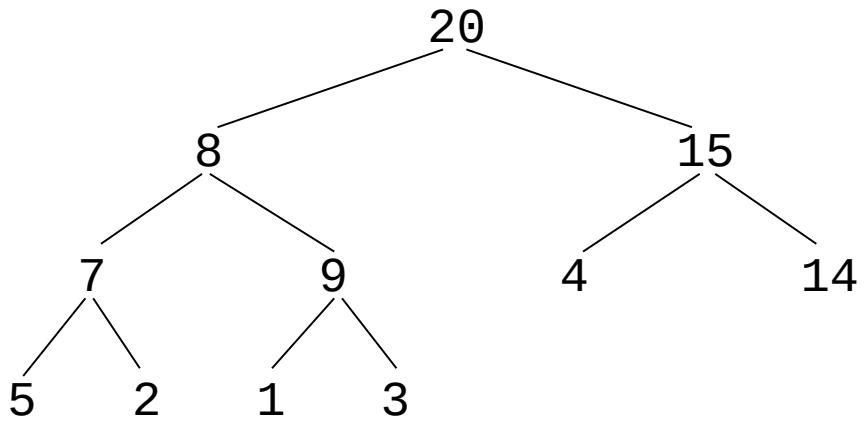
*If the candidates are seen randomly*

FR-67: **Heap Examples**




Valid Heap

FR-68: **Heap Examples**




Invalid Heap

FR-69: **Heap Operations**

- Assuming a Min-Heap
  - Insert
  - Delete Min
  - Decrease Key

FR-70: **Building a Heap**

Build a heap out of $n$ elements

- Start with an empty heap
- Do $n$ insertions into the heap

```
MaxHeap H = new MaxHeap();
for(i=0 < i<A.size(); i++)
   H.insert(A[i]);
```

Running time? $O(n \lg n)$ – is this bound tight?

FR-71: **Building a Heap** Total time: $c_1 + \sum_{i=1}^{n} c_2 \lg i$

FR-72: **Building a Heap** Total time: $c_1 + \sum_{i=1}^{n} c_2 \lg i$

$$\begin{aligned} c_1 + \sum_{i=1}^{n} c_2 \lg i &\geq \sum_{i=n/2}^{n} c_2 \lg i \\ &\geq \sum_{i=n/2}^{n} c_2 \lg(n/2) \\ &= (n/2)c_2 \lg(n/2) \\ &= (n/2)c_2((\lg n) - 1) \\ &\in \Omega(n \lg n) \end{aligned}$$

Running Time: $\Theta(n \lg n)$

FR-73: **Building a Heap**

Build a heap from the bottom up

- Place elements into a heap array
- Each leaf is a legal heap
- First potential problem is at location $\lfloor i/2 \rfloor$

FR-74: **Building a Heap**

Build a heap from the bottom up

- Place elements into a heap array
- Each leaf is a legal heap
- First potential problem is at location $\lfloor i/2 \rfloor$

```
for(i=n/2; i>=0; i--)
  siftdown(i);
```

FR-75: **Building a Heap**

How many swaps, worst case? If every `siftdown` has to swap all the way to a leaf:

| | |
|---|---|
| $n/4$ elements | 1 swap |
| $n/8$ elements | 2 swaps |
| $n/16$ elements | 3 swaps |
| $n/32$ elements | 4 swaps |
| ... | |

Total # of swaps:

$$n/4 + 2n/8 + 3n/16 + 4n/32 + \ldots + (\lg n)n/n$$

FR-76: **Heapsort**

- How can we use a heap to sort a list?
  - Build a heap out of the array we want to sort (Time $\Theta(n)$)
  - While the heap is not empty:
    - Remove the largest element
    - Place this element in the "empty space" just cleared by the deletion

Total time: $\Theta(n \lg n)$

FR-77: **Divide & Conquer**

- Quicksort:
  - Pick a pivot element
  - Divide the list into elements $<$ pivot, elements $>$ pivot
  - Recursively sort each of these two segments
  - No work required after recursive step
- Dividing the list is harder
- Combining solutions is easy (no real work required)

FR-78: **Quicksort**

```
Quicksort(A, low, high)
    if (low ¡ high) then
        pivotindex ← Partition(A, low, high)
        Quicksort(A, low, pivotindex − 1)
        Quicksort(A, pivotindex + 1, high)
```

FR-79: **Quicksort**

- How can we efficiently partition the list?

FR-80: **Quicksort**

- How can we efficiently partition the list?
- Method 1:
  - Maintain two indices, $i$ and $j$
  - Everything to left of $i \leq$ pivot
  - Everything to right if $j \geq$ pivot
  - Start $i$ at beginning of the list, $j$ at the end of the list, move them in maintaining the conditions above

FR-81: **Quicksort**

- How can we efficiently partition the list?
- Method 2:
  - Maintain two indices, $i$ and $j$
  - Everything to left of $i \leq$ pivot
  - Everything between $i$ and $j \geq$ pivot
  - Start both $i$ and $j$ at beginning of the list, increase them while maintaining the conditions above

FR-82: **Partition**

```
Partition(A, low, high)
    pivot = A[high]
    i ← low - 1
    for j ← low to high - 1 do
        if (A[j] ≤ pivot then
            i ← i + 1
            swap A[i] ↔ A[j]
    swap A[i+1] ↔ A[hight]
```

FR-83: **Partition**

Partition example:

5 7 1 3 6 2 8 4

FR-84: **Quicksort**

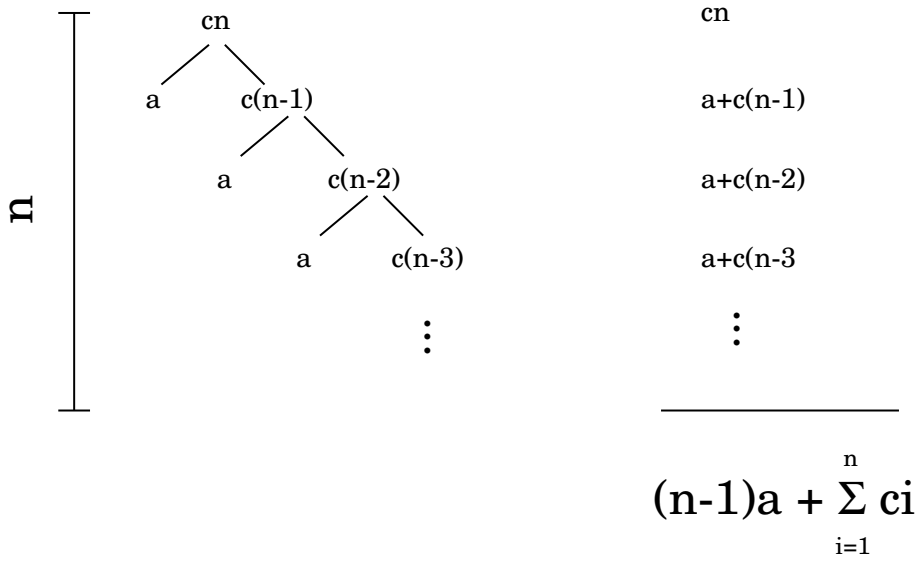
- Running time for Quicksort: Intuition

- Worst case: list is split into size 0, size (n-1)

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

Recursion Tree

FR-85: **Quicksort**




FR-86: **Quicksort**

Confirm $O(n^2)$ with substitution method:

$$T(n) = T(n-1) + c * n$$

FR-87: **Quicksort**

Confirm $O(n^2)$ with substitution method:

$$\begin{aligned} T(n) &= T(n-1) + c * n \\ &\leq c_1 * (n-1)^2 + c * n \\ &\leq c_1 * (n^2 - 2n + 1) + c * n \\ &\leq c_1 * n^2 + (c - 2 * c_1 + 1/n) * n \\ &\leq c_1 * n^2 \end{aligned}$$

(if $c_1 > (c + 1/n)/2$)

FR-88: **Quicksort**

Confirm $\Omega(n^2)$ with substitution method:

$$\begin{aligned} T(n) &= T(n-1) + c*n \\ &\geq c_1*(n-1)^2 + c*n \\ &\geq c_1*(n^2-2n+1) + c*n \\ &\geq c_1*(n^2-2n) + c*n \\ &\geq c_1*n^2 + (c-2*c_1)*n \\ &\geq c_1*n^2 \end{aligned}$$

(if $c_1 > c/2$)

FR-89: **Quicksort**

- Average case:
  - What is the average case?
  - We can *assume* that all permutations of the list are equally likely (is this a good assumption?)
  - What else can we do?

FR-90: **Partition**

```
Partition(A, low, high)
    pivot = A[high]
    i ← low - 1
    for j ← low to high - 1 do
        if (A[j] ≤ pivot then
            i ← i + 1
            swap A[i] ↔ A[j]
    swap A[i+1] ↔ A[hight]
```

FR-91: **Randomized Partition**

```
Partition(A, low, high)
    swap A[high] ↔ A[random(low,high)]
    pivot = A[high]
    i ← low - 1
    for j ← low to high - 1 do
        if (A[j] ≤ pivot then
            i ← i + 1
            swap A[i] ↔ A[j]
    swap A[i+1] ↔ A[hight]
```

FR-92: **Quicksort Analysis**

- OK, we can assume that all permutations are equally likely (especially if we randomize partition)
- How long does quicksort take in the average case?

FR-93: **Quicksort Analysis**

- Time for quicksort dominated by time spent in partition procedure
- Partition can be called a maximum of $n$ times (why)?
- Time for each call to partition is $\Theta(1)$ + # of times through for loop
- Total number of times the test (A[j] $\leq$ pivot) is done is proportional to the time spent for the loop
- Therefore, the total # of times the test (A[j] $\leq$ pivot) is a bound on the time for the entire algorithm

FR-94: **Quicksort Analysis**

Some definitions:

- Define $z_i$ to be the $i$th smallest element in the list
- Define $Z_{ij}$ to be the set of elements $z_i, z_{i+1}, \ldots z_j$

So, if our array $A$ = {3, 4, 1, 9, 10, 7} then:

- $z_1 = 1$, $z_2 = 3$, $z_3 = 4$, etc
- $Z_{35}$ = {4, 7, 9}
- $Z_{46}$ = {7, 9, 10}

FR-95: **Quicksort Analysis**

- Each pair of elements can be compared at most once (why)?
- Define an indicator variable $X_{ij} = I\{z_i \text{ is compared to } z_j\}$

$$
\begin{aligned}
X &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \\
E[X] &= E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}] \\
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ compared to } z_j\}
\end{aligned}
$$

FR-96: **Quicksort Analysis**

- Calculating $E[X_{ij}]$:
  - When will element $z_i$ be compared to $z_j$?
- A = {1, 2, 3, 4, 5, 6, 7, 8, 9 , 10}
- If pivot = 6
  - 6 will be compared to every other element
  - 1-5 will never be compared to anything in 7-10

FR-97: **Quicksort Analysis**

- Calculating $E[X_{ij}]$:
  - Given any two elements $z_i, z_j$, if we pick some element $x$ as a pivot such that $z_i < x < z_j$, then $z_i$ and $z_j$ will never be compared to each other
  - $z_i$ and $z_j$ will be compared with each other when the first element chosen $Z_{ij}$ is either $z_i$ or $z_j$

FR-98: **Quicksort Analysis**

$$\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first pivot selected from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is first from } Z_{ij}\} + \Pr\{z_j \text{ is first from } Z_{ij}\} \\
&= 1/(j-i+1) + 1/(j-i+1) \\
&= 2/(j-i+1)
\end{aligned}$$

FR-99: **Quicksort Analysis**

$$\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} \\
&< \sum_{i=1}^{n-1} 2\ln(n-i) + 1
\end{aligned}$$

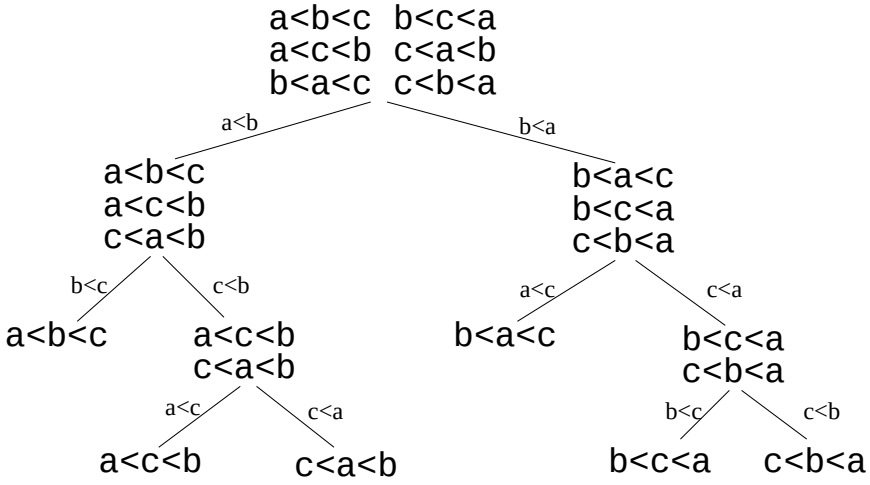
FR-100: **Quicksort Analysis**

$$\begin{aligned}
E[X] &< \sum_{i=1}^{n-1} 2\ln(n-i) + 1 \\
&< \sum_{i=1}^{n-1} 2\ln(n) + 1 \\
&< 2 * n\ln(n) + 1 \\
&\in O(n \lg n)
\end{aligned}$$

FR-101: **Comparison Sorting**

- Comparison sorts work by comparing elements
  - Can only compare 2 elements at a time
  - Check for $<$, $>$, $=$.
- All the sorts we have seen so far (Insertion, Quick, Merge, Heap, etc.) are comparison sorts
- If we know nothing about the list to be sorted, we need to use a comparison sort

FR-102: **Decision Trees** Insertion Sort on list $\{a, b, c\}$




FR-103: **Sorting Lower Bound**

- All comparison sorting algorithms can be represented by a decision tree with $n!$ leaves
- Worst-case number of comparisons required by a sorting algorithm represented by a decision tree is the height of the tree
- A decision tree with $n!$ leaves must have a height of at least $n \lg n$
- All comparison sorting algorithms have worst-case running time $\Omega(n \lg n)$

FR-104: **Counting Sort**

- Create the array $C[]$, such that $C[i] =$ # of times key $i$ appears in the array.
  - How?
- Modify $C[]$ such that $C[i] =$ the *index* of key $i$ in the sorted array. (assume no duplicate keys, for now)
- If $x \notin A$, we don't care about $C[x]$

FR-105: **Counting Sort Revisited**

- Create the array $C[]$, such that $C[i] =$ # of times key $i$ appears in the array.
- Modify $C[]$ such that $C[i] =$ the *index* of key $i$ in the sorted array. (assume no duplicate keys, for now)
- If $x \notin A$, we don't care about $C[x]$

```
for(i=1; i<C.length; i++)
  C[i] = C[i] + C[i-1];
```

- If the list has duplicates, then $C[i]$ will contain the index of the last occurence of $i$.
- Example: 3 1 2 3 9 8 7

FR-106: **Counting Sort**

```
for(i=0; i<A.length; i++)
   C[A[i].key()]++;
for(i=1; i<C.length; i++)
  C[i] = C[i] + C[i-1];

for (i=A.length - 1; i>=0; i++) {
   B[C[A[i].key()]] = A[i];
   C[A[i].key()]--;
}

for (i=0; i<A.length; i++)
   A[i] = B[i];
```

FR-107: **Radix Sort** Second Try:

- Sort by *least significant* digit first
- Then sort by next-least significant digit, using a Stable sort

  ...
- Sort by most significant digit, using a Stable sort

At the end, the list will be completely sorted. Why?

FR-108: **Selection Problem**

- What if we want to find the $k$th smallest element?
  - Median: $k = \lceil \frac{n}{2} \rceil$th smallest element, for odd $n$
- What is the obvious method?
- Can we do better?

FR-109: **Selection Problem**

```
Quicksort(A, low, high)
    if (low ¡ high) then
        pivotindex ← Partition(A, low, high)
        Quicksort(A, low, pivotindex 1)
        Quicksort(A, pivotindex + 1, high)
```

```
Partition(A, low, high)
    pivot = A[high]
    i ← low - 1
    for j ← low to high - 1 do
        if (A[j] ≤ pivot then
            i ← i + 1
            swap A[i] ↔ A[j]
    swap A[i+1] ↔ A[high]
```

FR-110: **Selection Problem**

```
Select(A,low,high,k)
    if (low = high)
        return A[low]
    pivot = Partition(A, low, high)
    adj_pivot = piv - low + 1
```

```
if (k = adj_pivot) then
    return A[pivot]
if (k ¡ adj_pivot) then
    return Select(A,low,pivot-1, k)
else
    return Select(A,pivot+1, high, pivot-adj_pivot)
```

Running time (Best and worst case)?

FR-111: **Selection Problem**

- Best case time:

$$T(n) = T(n/2) + c * n \in \Theta(n)$$

- Worst case time:

$$T(n) = T(n-1) + c * n \in \Theta(n^2)$$

- Average case time turns out to be $\Theta(n)$, but we'd like to get the worst-case time down.

FR-112: **Selection Problem**

- Improving worst-case time for selection
  - We need to guarantee a "good" pivot to get $\Theta(n)$ time for selection
  - How much time can we spend to find a good pivot, and still get $\Theta(n)$ time for selection?
  - $O(n)$ !

FR-113: **Selection Problem**

- Finding a "Good" pivot (one that is near the median) in linear time:
  - Split the list into $\frac{n}{5}$ list of length 5
  - Do an insertion sort on each of the $\frac{n}{5}$ lists to find the median of each of these lists
  - Call select recursively to find the median of the $\frac{n}{5}$ medians

FR-114: **Binary Search Trees**

- Binary Trees
- For each node n, (value stored at node n) $>$ (value stored in left subtree)
- For each node n, (value stored at node n) $<$ (value stored in right subtree)

FR-115: **Example Binary Search Trees**

- Examples:
  - Finding an element
  - Inserting an element
  - Deleting an element

FR-116: **Balanced BSTs**

- We can guarantee $O(\lg n)$ running time for insert/find/delete if we can guarantee the tree is balanced
- Several methods for guaranteeing a balanced tree
  - AVL trees & Red-Black trees are the most common
  - We'll look at Red-Black Trees

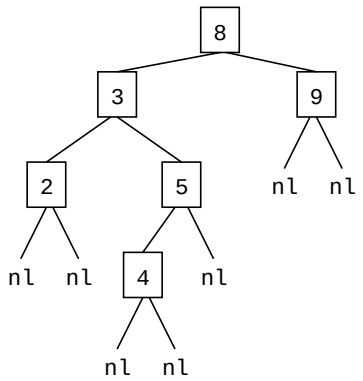
FR-117: **Red-Black Trees**

- Red-Black Trees as Binary Search trees, with "Null Leaves"
  - Examples of BSTs with "Null Leaves"
  - (Null leaves are mostly a notational convenience)

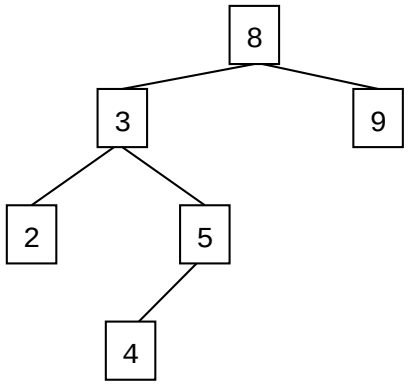
FR-118: **Red-Black Trees**

- Red-Black Trees are Binary Search trees, with "Null Leaves", and the following properties:
  - Every Node is either Red or Black
  - (Root is Black) <Not strictly required>
  - Each null "leaf" is Black
  - If a node is red, both children are black
  - For each node, all paths from the node to descendant leaves contain the same number of black nodes

FR-119: **Red-Black Trees**

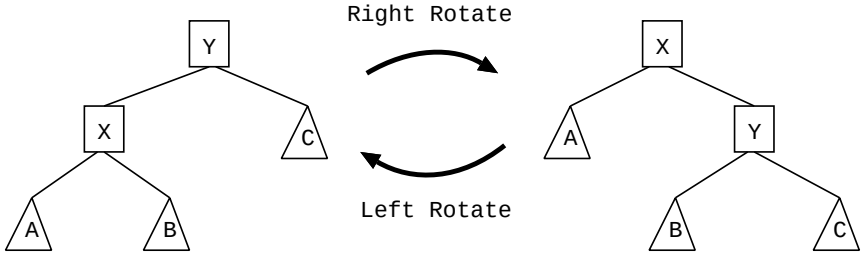



FR-120: **Red-Black Trees**



- Example Red-Black tree ("Null Leaves" left out for clarity)

FR-121: **Tree Rotations**




FR-122: **Tree Insertions**

- Always insert red nodes
- Which property would be violated by inserting a red node?

FR-123: **Tree Insertions**

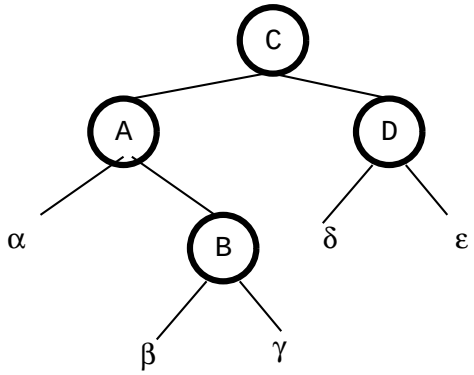
- Always insert red nodes
- Which property would be violated by inserting a red node?
  - Could have a red node with a red child
- Fix using tree rotations

FR-124: **Tree Insertions**

- To fix a red node with red child:
  - Case 1: Uncle is red
  - Case 2: Uncle is black, Inserted node is right child of parent, and parent is a left child of Grandparent (or node is left child, parent is right child)
  - Case 3: Uncle is black, Node is left child of parent, parent is left child of Grandparent (or node is righ child, parent is right child)

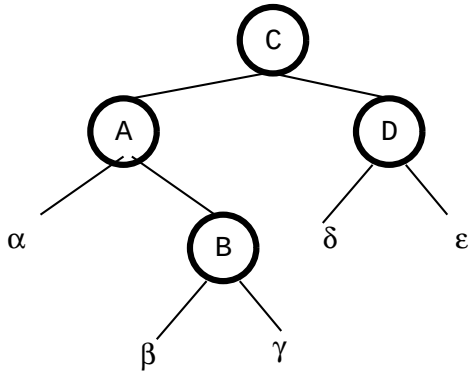
FR-125: **Case 1**

- Red Uncle



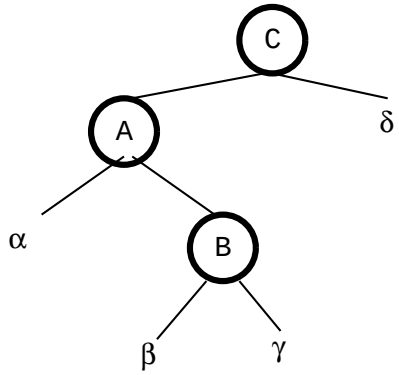

FR-126: **Case 1**

- Red Uncle



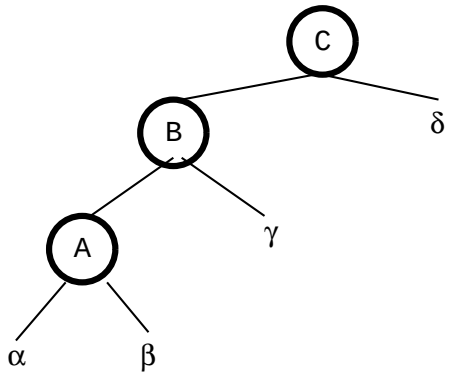

FR-127: **Case 2**

- Black Uncle / parent child different handedness



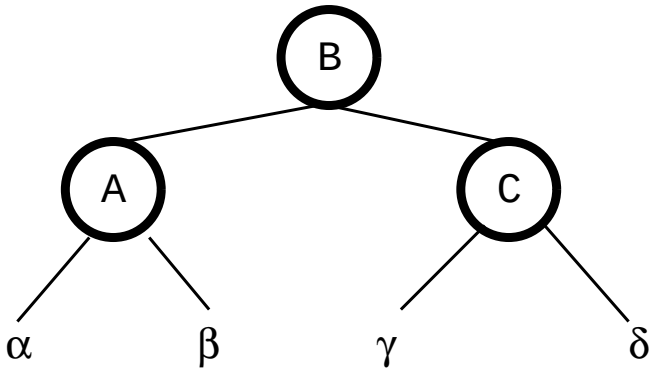

FR-128: **Case 3**

- Black Uncle / parent child same handedness




FR-129: **Case 3**

- Black Uncle / parent child same handedness




FR-130: **Deleting nodes**

- Deleting nodes
  - Delete nodes just like in standard BST
  - Which properties could be violated by deleting a red node?
    - Each node red or black
    - Black Root
    - Each red node has 2 black children
    - Black path length to leaves same for each node

FR-131: **Deleting nodes**

- Deleting nodes
  - Delete nodes just like in standard BST
  - Which properties could be violated by deleting a red node?
    - None!

FR-132: **Deleting Nodes**

- Deleting nodes
  - Delete nodes just like in standard BST
  - Which properties could be violated by deleting a black node?
    - Each node red or black
    - Black Root
    - Each red node has 2 black children
    - Black path length to leaves same for each node

FR-133: **Deleting Nodes**

- Deleting black node
  - If the child of the deleted node is red ... (show example on board)

FR-134: **Deleting Nodes**

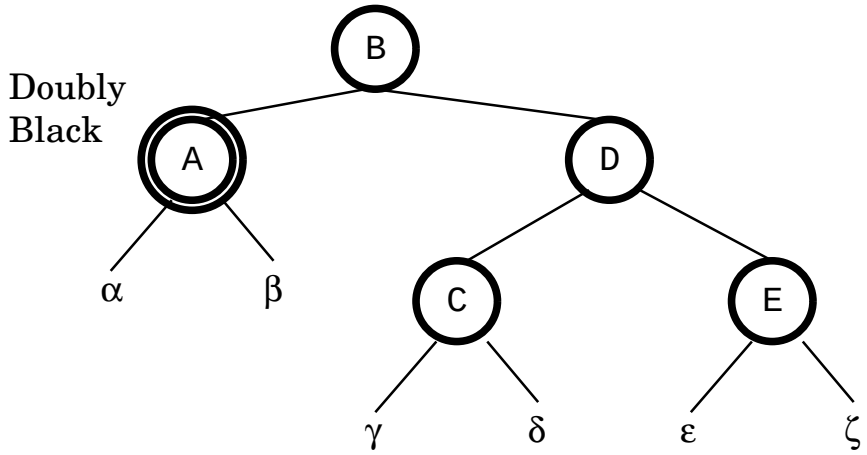
- Deleting black node

- If the child of the deleted node is black
  - Make the child "doubly black"
  - Push "extra blackness" up the tree until it can be removed by a rotation

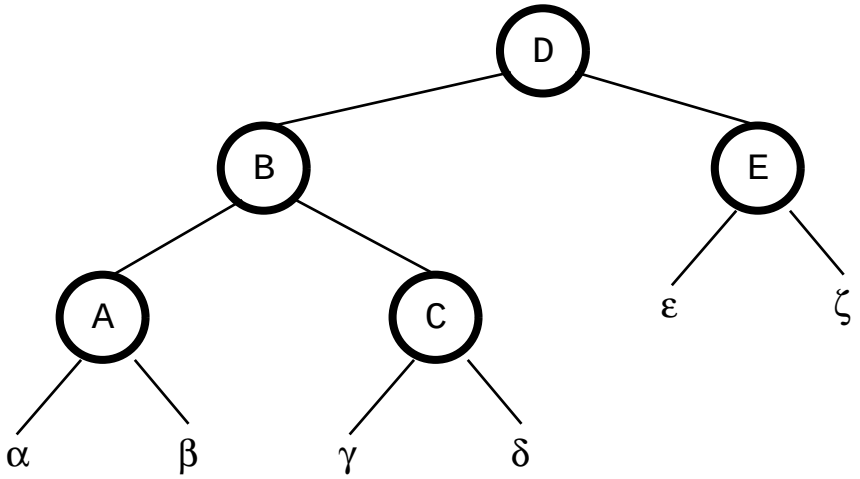
FR-135: **Deleting Nodes**

- X is "doubly black" node, X is a left child
  - Case 4:
    - X's sibling W is black, and W's right child is red
    - Can remove "double-blackness" of X with a single rotation

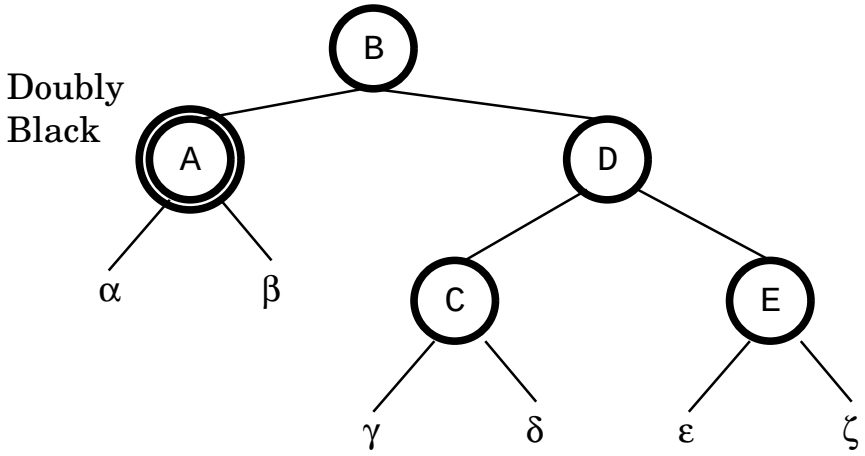
FR-136: **Deleting Nodes**



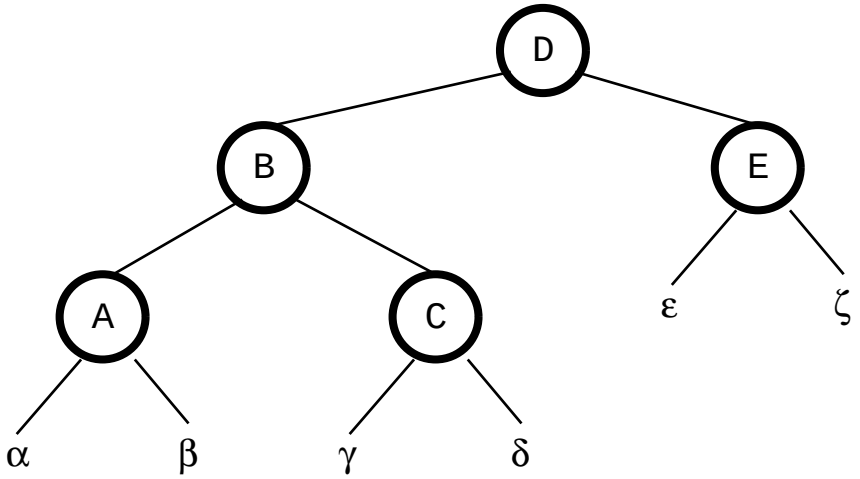

FR-137: **Deleting Nodes**




FR-138: **Deleting Nodes**



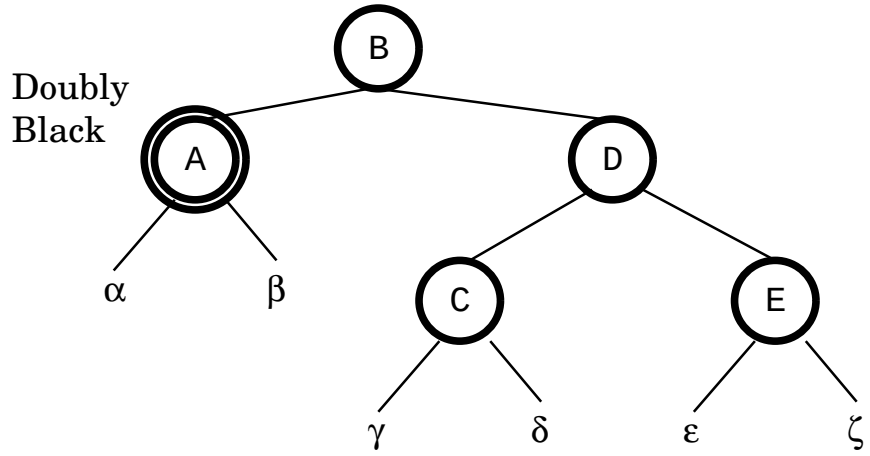

FR-139: **Deleting Nodes**



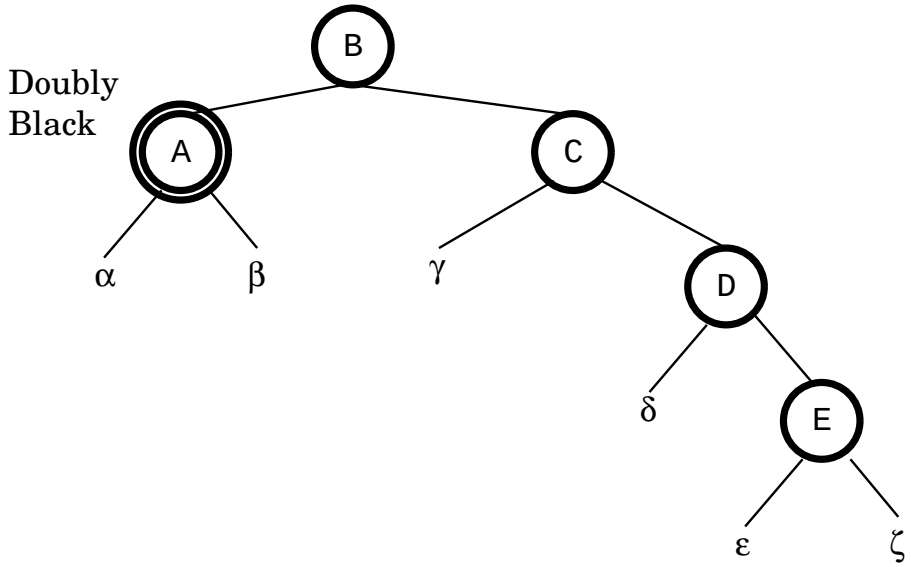

FR-140: **Deleting Nodes**

- X is “doubly black” node, X is a left child
  - Case 3:
    - X’s sibling W is black, and W’s left child is red, and right child is black
    - Single rotation to get to previous case

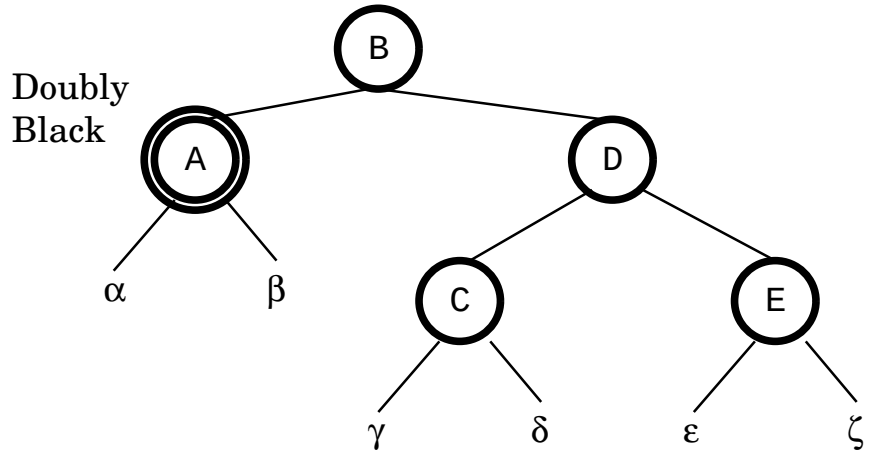
FR-141: **Deleting Nodes**



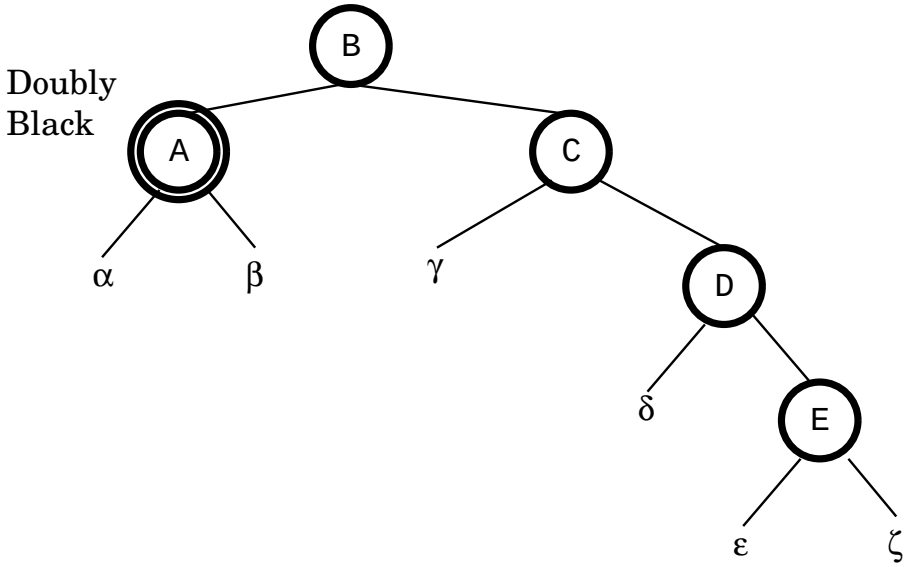

FR-142: **Deleting Nodes**




FR-143: **Deleting Nodes**



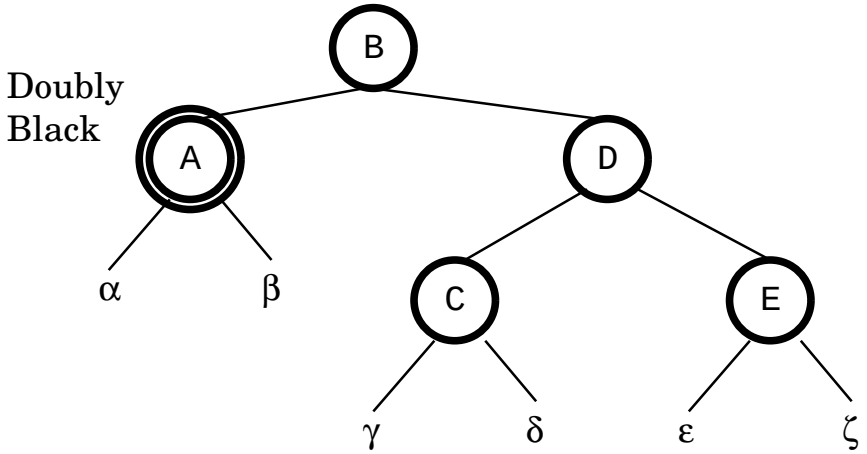

FR-144: **Deleting Nodes**



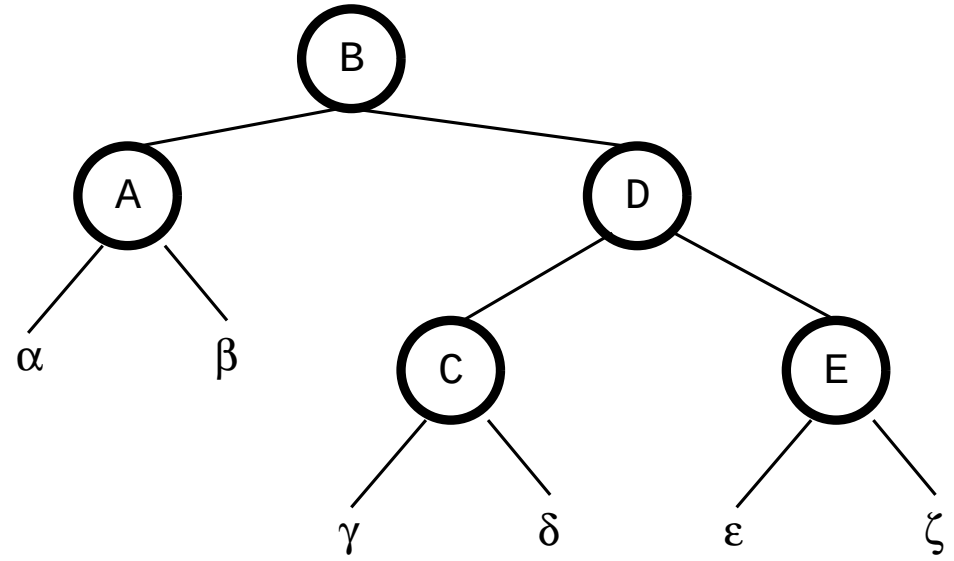

FR-145: **Deleting Nodes**

- X is "doubly black" node, X is a left child
  - Case 2:
    - X's sibling W is black, and both of W's children are black
    - Push "Blackness" of X and W to parent

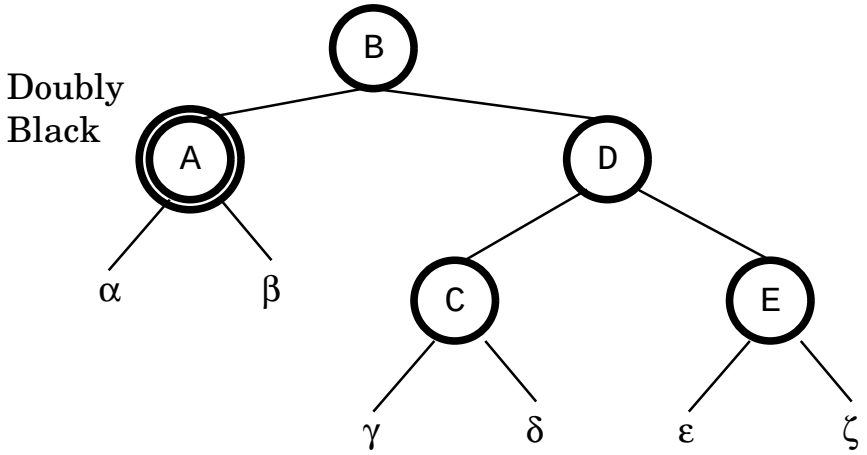
FR-146: **Deleting Nodes**



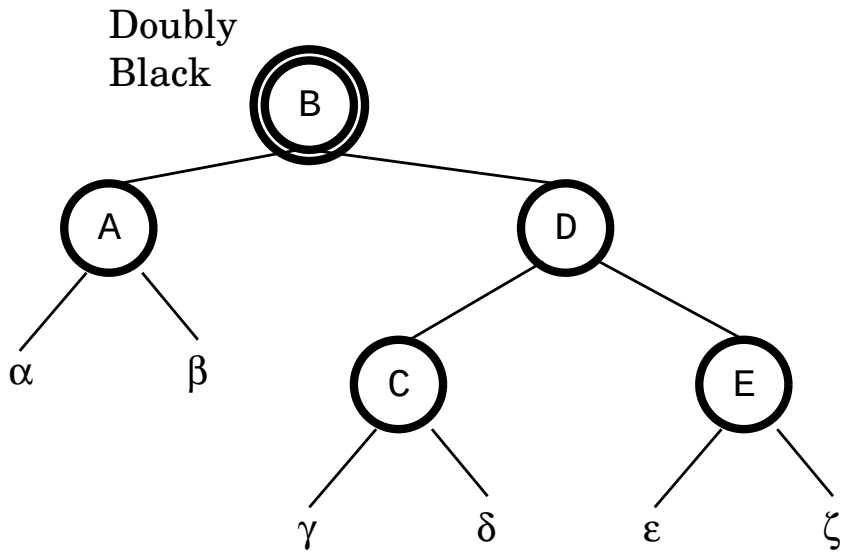

FR-147: **Deleting Nodes**




FR-148: **Deleting Nodes**



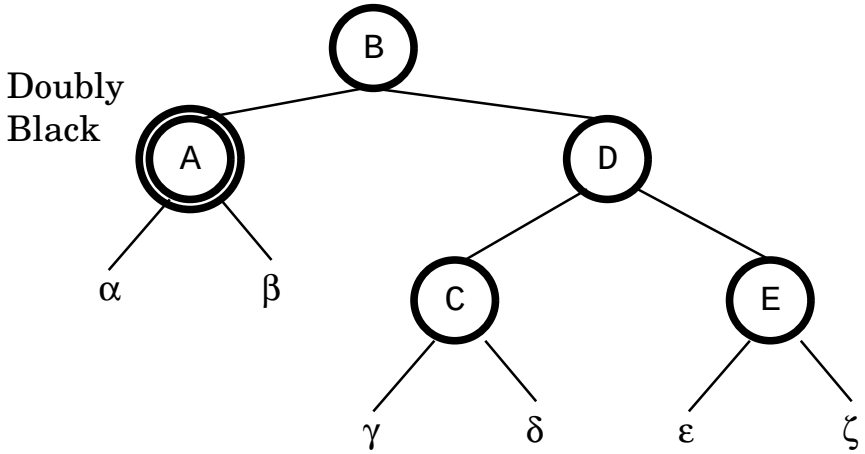

FR-149: **Deleting Nodes**



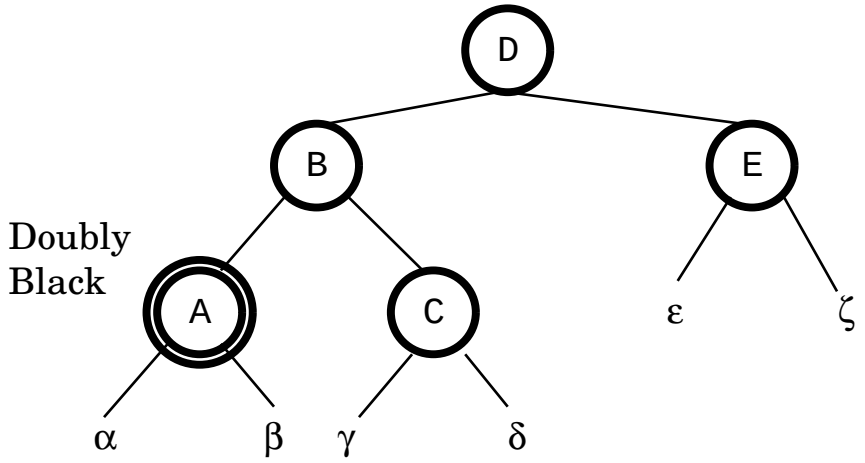

FR-150: **Deleting Nodes**

- X is "doubly black" node, X is a left child
  - Case 2:
    - X's sibling W is Red
    - Do a rotation, to make W black. Then one of the other cases will apply.

FR-151: **Deleting Nodes**




FR-152: **Deleting Nodes**




FR-153: **Deleting Nodes**

- Need to include symmetric cases
  - In all of the previous examples, swap left/right
  - (Go over at least one example)

FR-154: **Dynamic Order Statistics**

- Data structure with following operations:
  - Insert / Remove / Find $\Theta(\lg n)$
  - Find nth smallest element
  - Find the rank of any element (is it smallest, second smallest, etc)

How can we do this with red/black trees?

- How to find the rank of any element in a red/black tree
- How to find the $n$th element in a red/black tree

FR-155: **Dynamic Order Statistics**

- Addinging functionality to red/black trees
  - Finding the $n$th element in a red/black tree
  - Finding the rank of any element in a red/black tree
- What if we could add some other data to a red/black tree, to make this easier?
  - What should we add?
  - How could we use it?

FR-156: **Size Field**

- Add a "size" field to each node in a red/black tree
  - How can we use this size field to find the $n$th element in the tree?

- How can we use this size field to find the rank of any element?
- Can we maintain the size field, and still have insert/remove/find take time $O(\lg n)$?

FR-157: **Using Size Field**

- To find the $k$th element in a red/black tree with size field:
  - If # of elements in left subtree $= k - 1$, then root is the $k$th element
  - if # of elements in left subtree $> k - 1$, then the $k$th element is the $k$th element in the left subtree
  - if # of elements in the left subtree $= n < k - 1$, then the $k$th element is the $(n - (k + 1))$th element in the right subtree

FR-158: **Updating Size: Insert**

- Updating size field on insert:

FR-159: **Updating Size: Insert**

- Updating size field on insert:
  - As we go down the tree looking for the correct place to insert an element, add one to the size field of every node along the path from the root to the inserted element
    - (examples on board)

FR-160: **Updating Size: Delete**

- Updating size field on delete:

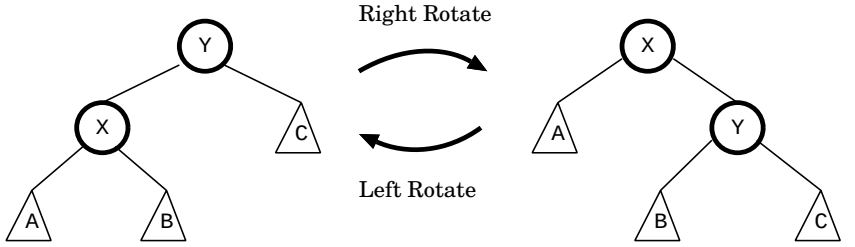
FR-161: **Updating Size: Delete**

- Updating size field on delete:
  - As we go down the tree looking for the element to delete, delete one from the size of evey node along the path from the root to the deleted element
    - (examples on board)
  - Need to be careful about trying to delete elements that aren't in the tree

FR-162: **Updating Size Field: Rotate**

- Updating size on rotations
  - How should sizes be updated on rotations?
    - (Picture on board)

FR-163: **Updating Size Field: Rotate**




Sizes of A,B,C not changed
Size(X) = Size(A) + Size(B) + 1
Size(Y) = Size(X) + Size(C) + 1

Sizes of A,B,C not changed
Size(Y) = Size(B) + Size(C) + 1
Size(X) = Size(A) + Size(Y) + 1

FR-164: **Augmenting Data Structures**

- Decide what extra information to add to each element of the data structure
- Make sure we can update this extra information for each operation on the data structure
- Add operations that use this extra information
  - New operations
  - Do old operations more efficiently

(Finding rank example)

FR-165: **Augmenting Data Structures**

- For Red/Black trees:
  - If extra information in a node is dependent only on the node itself, and values in left & right children
  - Then, we can always update this information during insert and delete in time $O(\lg n)$

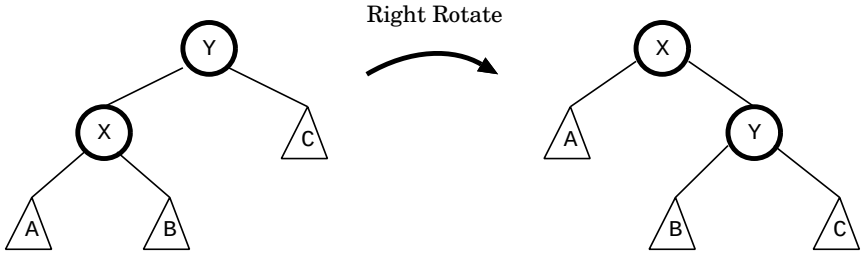
FR-166: **Augmenting Data Structures**

- On an insert:
  - Add the leaf
  - Update information on path from leaf to root after the insertion
    - Extra time: $O(\lg n)$
  - Rotate as necessary

FR-167: **Augmenting Data Structures**

- On a delete:
  - Delete the node
  - Update information on path from deleted node to root after deletion is completed
  - (also works for deletion of node w/ 2 children, do example)
    - Extra time: $O(\lg n)$
  - Rotate as necessary

FR-168: **Augmenting Data Structures**




- Values in A,B,C don't need to change
- Values in X,Y can be changed by looking at A,B,C
- Might need to propagate change up the tree (time $O(\lg n)$)

FR-169: **Dynamic Programming**

- Simple, recursive solution to a problem
- Straightforward implementation of recursion leads to exponential behavior, because of repeated subproblems
- Create a table of solutions to subproblems
- Fill in the table, in an order that guarantees that each time you need to fill in a square, the values of the required subproblems have already been filled in

FR-170: **Dynamic Programming**

- To solve a Dynamic Programming problem:
  - Define the table
    - How many dimensions?
    - What does each entry mean?
  - Give a formula for filling in table entries
    - Recursive and base cases
  - Give an ordering to fill in the table

FR-171: **Dynamic Programming**

Give a $O(n^2)$ algorithm for finding the longest monotonically increasing subsequence of a sequence of numbers

FR-172: **Dynamic Programming**

- $L[i]$ = length of longest monotonically increasing subsequence of elements $a_1 \ldots a_i$ *that contains element* $a_i$
  - $L[1] = 1$
  - $L[i] =$
    - 1, if If $a_i$ is the smallest value in $a_1 \ldots a_{i-1}$
    - $1 + L[k]$, where $L[k]$ is the largest value in $a_i \ldots a_{i-1}$ such that $a_k < a_i$

FR-173: **Greedy Algorithms**

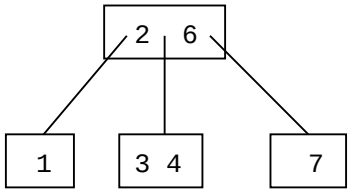
- Often (but not always!), creating a greedy solution is easy
- Often (but not always!) takes time $\Theta(n \lg n)$ (why?)
- Hard part is proving that the algorithm is correct

FR-174: **Proving Greedy**

- To prove a greedy algorithm is correct:
  - Greedy Choice
    - At least one optimal solution contains the greedy choice
  - Optimal Substructure
    - An optimal solution can be made from the greedy choice plus an optimal solution to the remaining subproblem
- Why is this enough?

FR-175: **B-Trees**

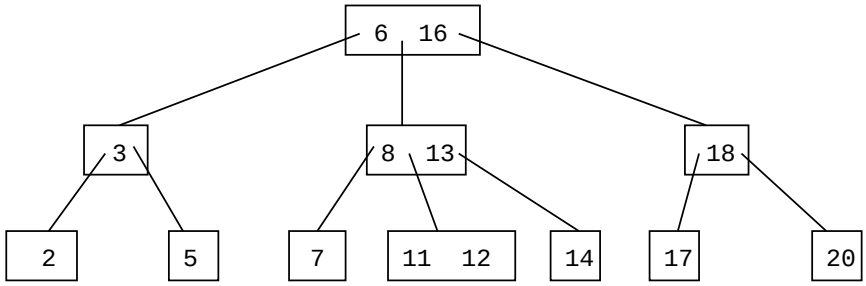
- Generalized Binary Search Trees
  - Each node can store several keys, instead of just one
  - Values in subtrees between values in surrounding keys
  - For non leaves, # of children = # of keys + 1




FR-176: **2-3 Trees**

- Generalized Binary Search Tree
  - Each node has 1 or 2 keys
  - Each (non-leaf) node has 2-3 children
    - hence the name, 2-3 Trees
  - All leaves are at the same depth

FR-177: **Example 2-3 Tree**




FR-178: **Example 2-3 Tree**

- Finding an element in a 2-3 tree
- Inserting an element into a 2-3 tree
- Deleting an element from a 2-3 tree

FR-179: **B-Trees**

- A B-Tree of maximum degree k:
  - All interior nodes have $\lceil k/2 \rceil \ldots k$ children
  - All nodes have $\lceil k/2 \rceil - 1 \ldots k - 1$ keys
- 2-3 Tree is a B-Tree of maximum degree 3

FR-180: **B-Trees**

- Preemptive Splitting
  - If the maximum degree is even, we can implement an insert with a single pass down the tree (instead of a pass down, and then a pass up to clean up)
  - When inserting into any subtree tree, if the root of that tree is full, split the root before inserting
    - Every time we want to do a split, we know our parent is not full.

(examples, use visualization) FR-181: **Amortized Analysis**

- Doing $n$ operations on a data structure
- Some operations take longer than others
- Only care about the average time for an operation
  - Be doing many operations on the data structure over the course of an algorithm
  - Care about the total time for all operations

FR-182: **Aggregate Method**

- Aggregate method
  - Total cost for $n$ operations is $g(n)$
  - Amortized cost for 1 operation is $\frac{g(n)}{n}$
- Works well with just a single operation
- Can be trickier with different kinds of operations
  - Insert & find & delete
  - Need to either modle the order the operations will be applied, or prove that a particular order maximizes the total running time

FR-183: **Accounting Method**

- Accounting Method
  - Assign a cost for each operation
    - Called "amortized cost"
  - When amortized cost $>$ actual cost, create a "credit" which can be used when actual cost $>$ amortized cost
  - Must design costs so that all sequences of operations always leave a "positive account"

FR-184: **Potential Method**

- Definte a "potential" for data structures that your algorithm uses
  - Kind of like potential energy
- When the amortized cost is greater than the actual cost, increase the potential of the data structure
- When the amortize cost is less than the actual cost, decrease the potential of the data structure
  - Potential can never be negative

FR-185: **Potential Method**

- The potential function is on the *Data Structure*, not the operations
- Don't talk about the potential of a push or a pop
- Instead, talk about the potential of the stack
  - Define a potential function on the data structure
  - Use the potential function and actual cost to determine amortized cost

FR-186: **Binomial Heaps**

- Binomial Trees
- Binomial Heap
  - Set of binomial trees, sorted by degree
  - No two trees have the same degree
  - Each tree has heap property
  - Merging two binomial heaps, deleting minimum element

FR-187: **Fibonacci Heaps**

- A Fibonacci Heap, like a Binomial Heap, is a collection of min-heap ordered trees
  - No restriction on the # of trees of the same size
  - (Each tree could be "not really" a binomial tree, after decrease key operations)
- Maintain a pointer to tree with smallest root

FR-188: **Fibonacci Heaps**

- Merging heaps
- Finding minimum
- Removing minimum
  - Consolidate root list
- Decrease key

FR-189: **Disjoint Sets**

- Maintain a collection of sets
- Operations:
  - Determine which set an element is in
  - Union (merge) two sets
- Initially, each element is in its own set
  - # of sets = # of elements

FR-190: **Disjoint Sets**

- Represetning disjoint sets
  - Using trees
- Union-by-Rank
- Path Compression

FR-191: **Graphs**

- Graph Representation
  - Adjacency list
  - Adjacency Matrix
- Relative advantages of each representation?

FR-192: **Graphs**

- Graph Traversals
  - BFS
  - DFS
    - Discovery / Finish Times
    - Topological Sort
    - Connected Components

FR-193: **Minimal Cost Spanning Tree**

- Minimal Cost Spanning Tree
  - Given a weighted, undirected graph $G$
  - Spanning tree of $G$ which minimizes the sum of all weights on edges of spanning tree

FR-194: **Calculating MST**

- Generic MST algorithm:

$A \leftarrow \{\}$
while $A$ does not form a spanning tree
  find an edge $(u, v)$ that is safe for $A$
  $A \leftarrow A \cup \{(u, v)\}$

- $(u, v)$ is safe to for $A$ when $A \cup \{(u, v)\}$ is a subset of some MST

FR-195: **Calculating MST**

- Correctness of MST algorithms
  - Cuts that respects edges
  - Edges crossing cut
  - Light edges

FR-196: **MST Algorithms**

- Kruskal
  - Combine trees in forrest into one tree
- Prim
  - Start from initial vertex
  - Build tree outward

FR-197: **Single Source Shortest Path**

- BFS
  - Works for uniform cost edges
- Dijkstra's Algorithm
  - *Remarkably* similar to Prim's MST algorith
  - Doesn't work with negative edges

FR-198: **Dijkstra Code**

```
void Dijkstra(Edge G[], int s, tableEntry T[]) {
  int i, v;
  Edge e;
  for(i=0; i<G.length; i++) {
    T[i].distance = Integer.MAX_VALUE;
    T[i].path = -1;
    T[i].known = false;
  }
  T[s].distance = 0;
  for (i=0; i < G.length; i++) {
    v = minUnknownVertex(T);
    T[v].known = true;
    for (e = G[v]; e != null; e = e.next) {
      if (T[e.neighbor].distance >
            T[v].distance + e.cost) {
        T[e.neighbor].distance = T[v].distance + e.cost;
        T[e.neighbor].path = v;
      }
    }
  }
}
```

FR-199: **Dijkstra Running Time**

- If minUnknownVertex(T) is calculated by doing a linear search through the table:
  - Each minUnknownVertex call takes time $\Theta(|V|)$
    - Called $|V|$ times – total time for all calls to minUnkownVertex: $\Theta(|V|^2)$
  - If statement is executed $|E|$ times, each time takes time $O(1)$
  - Total time: $O(|V|^2 + |E|) = O(|V|^2)$.

FR-200: **Dijkstra Running Time**

- If minUnknownVertex(T) is calculated by inserting all vertices into a min-heap (using distances as key) updating the heap as the distances are changed
  - Each minUnknownVertex call tatkes time $\Theta(\lg |V|)$
    - Called $|V|$ times – total time for all calls to minUnknownVertex: $\Theta(|V| \lg |V|)$
  - If statement is executed $|E|$ times – each time takes time $O(\lg |V|)$, since we need to update (decrement) keys in heap

- Total time: $O(|V| \lg |V| + |E| \lg |V|) \in O(|E| \lg |V|)$

FR-201: **Dijkstra Running Time**

- If minUnknownVertex(T) is calculated by inserting all vertices into a Fibonacci heap (using distances as key) updating the heap as the distances are changed
  - Each minUnknownVertex call takes amortized time $\Theta(\lg |V|)$
    - Called $|V|$ times – total amortized time for all calls to minUnknownVertex: $\Theta(|V| \lg |V|)$
  - If statement is executed $|E|$ times – each time takes amortized time $O(1)$, since decrementing keys takes time $O(1)$.
  - Total time: $O(|V| \lg |V| + |E|)$

FR-202: **Bellman-Ford**

- For each node $v$, maintiain:
  - A "distance estimate" from source to $v$, $d[v]$
  - Parent of $v$, $\pi[v]$, that gives this distance estimate
- Start with $d[v] = \infty$, $\pi[v]$ = nil for all nodes
- Set $d[\text{source}] = 0$
- udpate estimates by "relaxing" edges

FR-203: **Bellman-Ford**

- Relax all edges edges in the graph (in any order)
- Repeat until relax steps cause no change
  - After first relaxing, all optimal paths from source of length 1 are computed
  - After second relaxing, all optimal paths from source of length 2 are computed
  - after $|V| - 1$ relaxing, all optimal paths of length $|V| - 1$ are computed
  - If some path of length $|V|$ is cheaper than a path of length $|V| - 1$ that means ...
    - Negative weight cycle

FR-204: **Bellman-Ford**

```
BellamanFord(G, s)
    Initialize d[], π[]
    for i ← 1 to |V| − 1 do
        for each edge (u, v) ∈ G do
            if d[v] > d[u] + w(u, v)
                d[v] ← d[u] + w(u, v)
                π[v] ← u
    for each edge (u, v) ∈ G do
        if d[v] > d[u] + w(u, v)
            return false
    return true
```

FR-205: **Bellman-Ford**

- Running time:
  - Each iteration requires us to relax all $|E|$ edges
  - Each single relaxation takes time $O(1)$
  - $|V| - 1$ iterations ($|V|$ if we are checking for negative weight cycles)
  - Total running time $O(|V| * |E|)$

FR-206: **All-Source Shortest Path**

- What if we want to find the shortest path from all vertices to all other vertices?
- How can we do it?
  - Run Dijktra's Algorithm $V$ times
  - How long will this take?
  - $\Theta(V^2 \lg V + VE)$ (using Fibonacci heaps)
    - Doesn't work if there are negative edges! Running Bellman-Ford $V$ times (which does work with negative edges) takes time $O(V^2E)$ – which is $\Theta(V^4)$ for dense graphs

FR-207: **Floyd's Algorithm**

- Vertices numbered from 0..n
- $k$-path from vertex $v$ to vertex $u$ is a path whose intermediate vertices (other than $v$ and $u$) contain only vertices numbered $k$ or less
- 0-path is a direct link

FR-208: **Floyd's Algorithm**

- Let $D_k[v, w]$ be the length of the shortest $k$-path from $v$ to $w$.
- $D_0[v, w]$ = cost of arc from $v$ to $w$ ($\infty$ if no direct link)
- $D_k[v, w]$ = MIN($D_{k-1}[v, w]$, $D_{k-1}[v, k] + D_{k-1}[k, w]$)
- Create $D_0$, use $D_0$ to create $D_1$, use $D_1$ to create $D_2$, and so on – until we have $D_n$

FR-209: **Floyd's Algorithm**

```
Floyd(Edge G[], int D[][]) {
  int i,j,k

  Initialize D, D[i][j] = cost from i to j

  for (k=0; k<G.length; k++;
    for(i=0; i<G.length; i++)
      for(j=0; j<G.length; j++)
        if ((D[i][k] != Integer.MAX_VALUE) &&
            (D[k][j] != Integer.MAX_VALUE) &&
            (D[i][j] > (D[i,k] + D[k,j])))
          D[i][j] = D[i][k] + D[k][j]
}
```

FR-210: **Johnson's Algorithm**

- Yet another all-pairs shortest path algorithm
- Time $O(|V|^2 \lg |V| + |V| * |E|)$
  - If graph is dense ($|E| \in \Theta(|V|^2)$) , no better than Floyd
  - If graph is sparse, better than Floyd
- Basic Idea: Run Dijkstra $|V|$ times
  - Need to modify graph to remove negative edges

FR-211: **Johnson's Algorithm**

Johnson($G$)
Add $s$ to $G$, with 0 weight edges to all vertices
if Bellman-Ford($G, s$) = FALSE
There is a negative weight cycle, fail
for each vertex $v \in G$
set $h(v) \leftarrow \delta(s, v)$ from B-F
for each edge $(u, v) \in G$
$\hat{w}(u, v) = w(u, v) + h(u) - h(v)$
for each vertex $u \in G$
run Dijkstra($G, \hat{w}, u$) to compute $\hat{\delta}(u, v)$
$\delta(u, v) = \hat{\delta}(u, v) + h(v) - h(u)$

FR-212: **Flow Networks**

- Directed Graph $G$
- Each edge weigh is a "capacity"
  - Amount of water/second that can flow through a pipe, for instance
- Single source $S$, single sink $t$
- Calculate maximum flow through graph

FR-213: **Flow Networks**

- Flow: Function: $V \times V \rightarrow R$
  - Flow from each vertex to every other vertex
  - $f(u, v)$ is the direct flow from $u$ to $v$
- Properties:
  - $\forall u, v \in V, f(u, v) \leq c(u, v)$
  - $\forall u, v \in V, f(u, v) = -f(v, u)$
  - $\forall u, v \in V - \{s, t\}, \sum_{v \in V} f(u, v) = 0$
- Total flow, $|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$

FR-214: **Flow Networks**

- Single Source / Single Sink
  - Assume that there is always a single source and a single sink
  - Don't lost any expressive power – always transform a problem with multiple sources and multiple sinks to an equivalent problem with a single source and a single sink
  - How?

FR-215: **Flow Networks**

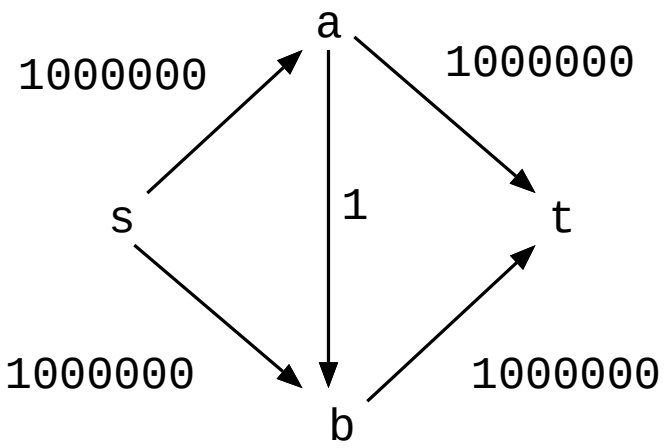
- Residual capacity
  - $c_f(u, v)$ is the residual capacity of edge $(u, v)$
  - $c_f(u, v) = c(u, v) - f(u, v)$
  - Note that it is possible for the residual capacity of an edge to be greater than the total capacity
    - Cancelling flow in the opposite direction

FR-216: **Ford-Fulkerson Method**

Ford-Fulkerson($G, s, t$)
    initialize flow $f$ to 0
    while there is an augmenting path $p$
        augment flow $f$ along $p$
    return $f$

FR-217: **Ford-Fulkerson Method**

- Could take as many as $|f|$ iterations:




FR-218: **Edmonds-Karp Algorithm**

- How can we be smart about choosing the augmenting path, to avoid the previous case?
  - We can get better performance by always picking the shortest path (path with the fewest edges)

- We can quickly find the shortest path by doing a BFS from the source in the residual network, to find the shortest augmenting path
- If we always choose the shortest augmenting path (i.e., smallest number of edges), total number of iterations is $O(|V| * |E|)$, for a total running time of $O(|V| * |E|^2)$

FR-219: **Push-Relabel Algorithms**

- New algorithm for calculating maximum flow
- Basic idea:
  - Allow vertices to be "overfull" (have more inflow than outflow)
  - Push full capacity out of edges from source
  - Push overflow at each vertex forward to the sink
  - Push excess flow back to source

FR-220: **Push-Relabel Algorithms**

Push($u, v$)
  Applies when:
    $u$ is overflowing
    $c_f(u, v) > 0$
    $h[u] = h[v] + 1$
  Action:
    Push $\min$(overflow$[u]$,$c_f(u, v)$) to $v$

FR-221: **Push-Relabel Algorithms**

Relabel(u)
  Applies when:
    $u$ is overflowing
    For all $v$ such that $c_f(u, v) > 0$
      $h[v] \geq h[u]$
  Action:
    $h[u] \leftarrow h[u] + 1$

FR-222: **Push-Relabel Algorithms**

Push-Relabel($G$)
  Initialize-Preflow($G, s$)
  while there exists an applicable push/relabel
    implement push/relabel

FR-223: **Push-Relabel Algorithms**

Push-Relabel($G$)
  Initialize-Preflow($G, s$)
  while there exists an applicable push/relabel
    implement push/relabel

- Pick the operations (push/relabel) arbitrarily, time is $O(|V|^2E)$
  - (We won't prove this result, though the proof is in the book)
- Can do better with relabel-to-front
  - Specific ordering for doing push-relabel
  - Time $O(|V|^3)$, also not proven here, proof in text

FR-224: **Cross Products**

- Given any two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$
  - Cross Product: $p_1 \times p_2 = x_1y_2 - x_2y_1$

$$\begin{aligned} p_1 \times p_2 &= x_1y_2 - x_2y_1 \\ &= -1 * (x_2y_1 - x_1y_2) \\ &= -p_2 \times p_1 \end{aligned}$$

FR-225: **Cross Products**

- Given two line segments $\overline{p_0p_1}$ and $\overline{p_1p_2}$, which direction does angle $\angle p_0p_1p_2$ turn?
  - $(p_2 - p_0) \times (p_1 - p_0)$ is positive, left turn
  - $(p_2 - p_0) \times (p_1 - p_0)$ is negative, right turn
  - $(p_2 - p_0) \times (p_1 - p_0)$ is zero, no turn (colinear)

FR-226: **Line Segment Intersection**

- Given two line segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$, do they intersect?
  - Each segment straddles the line containing the other
  - An endpoint of one segment lies on the other segment

FR-227: **Line Segment Intersection**

- How can we determine if $p_3$ is on the segment $\overline{p_1p_2}$?
  - $p_3$ is on the line defined by $p_1$ and $p_2$
    - $(p_2 - p_1) \times (p_3 - p_1) = 0$
  - $p_3$ is in the proper range along that line
    - $p_{3_x} \geq p_{1_x} \&\& p_{3_x} \leq p_{2_x}$ or $p_{3_x} \leq p_{1_x} \&\& p_{3_x} \geq p_{2_x}$
    - $p_{3_y} \geq p_{1_y} \&\& p_{3_y} \leq p_{2_y}$ or $p_{3_y} \leq p_{1_y} \&\& p_{3_y} \geq p_{2_y}$

FR-228: **Convex Hull**

- Given a set of points, what is the smallest convex polygon that contains all points

- Alternately, if all of the points were nails in a board, and we placed a rubber band around all of them, what shape would it form?

FR-229: **Convex Hull**

- Graham's Scan Algorithm
  - Go through all the points in order
  - Push points onto a stack
  - Pop off points that don't form part of the convex hull
  - When we're done, stack contains the points in the convex hull

FR-230: **Convex Hull**

- Different Convex Hull algorithm
- Idea:
  - Attach a string to the lowest point
  - Rotate string counterclockwise, unti it hits a point – this point is in the Convex Hull
  - Keep going until the highest point is reached
  - Continue around back to initial point

FR-231: **Closest Pair of Points**

- Divide & Conquer
  - Divide the list points in half (by a vertical line)
  - Recursively determine the closest pair in each half
  - Smallest distance between points is the minimum of:
    - Smallest distance in left half of points
    - Smallest distance in right half of points
    - Smallest distance that crosses from left to right

FR-232: **String Matching**

- Given a source text, and a string to match, where does the string appear in the text?
- Example: ababbabbaba and abbab

| a | b | a | b | b | a | b | b | a | b | a |
|---|---|---|---|---|---|---|---|---|---|---|
| | | x | x | x | x | x | | | | |
| | | | | | x | x | x | x | x | |

FR-233: **String Matching**

NAIVE-STRING-MATCHER($T$, $P$)
```
n ← length[T]
m ← length[P]
for s ← 0 to n − m do
    match ← false
    for j ← 1 to m do
        if T[i + j] ≠ T[j] then
            match ← false
    if match then
        Print "Pattern occurs with shift" s
```

FR-234: **Rabin-Karp**

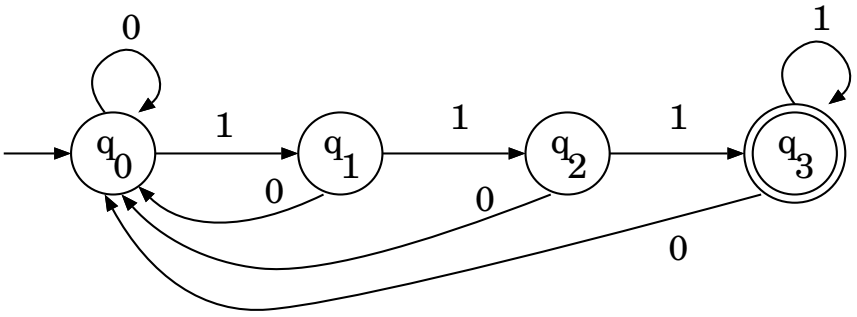
- Convert pattern to integer
  - Use modular arithmetic to make the number size managable
- Check for matches against this integer
  - Each hit might be spurious, need to verify
- Every time we get a potential hit, check the actual strings

FR-235: **DFA**

- Start in the initial state
- Go through the string, one character at a time, until the string is exhausted
- Determine if we are in a final state at the end of the string
  - If so, string is accepted
  - If not, string is rejected

FR-236: **DFA**

- All strings over $\{0,1\}$ that end in 111




FR-237: **DFA**

- You can use the DFA for all strings that end in 1001 to find all occurrences of the substring 1001 in a larger string
  - Start at the beginning if the larger string, in state $q_0$
  - Go through the string one symbol at a time, moving through the DFA
  - Every time we enter a final state, that's a match

FR-238: **DFA**

- Creating transition function $\delta$:
- Create a new concept: $\sigma_P(x)$
  - Length of the longest prefix of $P$ that is a suffix of $x$
  - $P = aba$

  - $\sigma_P(cba) = 1, \sigma_P(abc) = 0, \sigma_P(cab) = 2, \sigma_P(caba) = 3$
- $P_k$ = first $k$ symbols of $P$

FR-239: **DFA**

- $\delta(q, a) = \sigma(P_q a)$
- To find $\delta(q, a)$:
  - Start with string $P_q$: first $q$ characters of $P$
  - Append $a$, to get $P_q a$
  - Find the longest prefix of $P$ that is a suffix of $P_q a$.

FR-240: **DFA**

- Building $\delta$:

$m \leftarrow$ length[$P$]
for $q \leftarrow 0$ to $m$ do
  for each character $a \in \Sigma$ do
    $k \leftarrow \min(m + 1, q + 2)$
    do
      $k \leftarrow k - 1$
    until $P_k$ =] $P_q a$
    $\delta(q, a) \leftarrow k$

FR-241: **Knuth-Morris-Pratt**

- Maximum overlap array
  - How much can the string overlap with itself at each position?

| a | b | a | b | a | a | b | b | a |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 1 | 2 | 0 | 1 |

FR-242: **Knuth-Morris-Pratt**

- Prefix Function $\pi$:
  - $\pi[q] = max\{k : k < q \&\& P_k$ =] $P_q\}$
  - $\pi[q]$ is the length of the longest prefix of $P$ that is a proper suffix of $P_q$

FR-243: **Knuth-Morris-Pratt**

- Try to match pattern to input
- When a mismatch occurs, the $\pi$ array tells us how far to shift the pattern forward

Pattern: ababb $\pi$:

| a | b | a | b | b |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 0 |

Input String:

| a | b | a | b | a | b | b | a | b | b | a | b | a | b | a | b | b |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | a | b | b | | | | | | | | | | | | |

FR-244: **Knuth-Morris-Pratt**

- Creating $\pi$ array

```
m ← length[P]
π[1] ← 0
k ← 0
for q ← 2 to m do
    while k > 0 and P[k + 1] ≠ P[q]
        k ← π[k]
    if P[k + 1] = P[q]
        k ← k + 1
    π[q] ← k
```

FR-245: **Knuth-Morris-Pratt**

```
KMP-Matching(T, P)
    m ← length[P]
    n ← length[T]
    π ← ComputePI(P)
    q ← 0
    for i ← 1 to n do
        while q > 0 and P[q + 1] ≠ T[i]
            q ← π[q]
        if P[q + 1] = T[i]
            q ← q + 1
        if q = m
            Print "Match found at" i − m
            q ← π[q]
```

FR-246: **Classes of Problems**

- Consider three problem classes:
  - Polynomial (P)
  - Nondeterminisitic Polynomial (NP)
  - NP-Complete
- (only scratch the surface, take Automata Theory to go in depth)

FR-247: **Class P**

- Given a problem, we can find a solution in polynomial time
  - Time is polynomial in the length of the problem description
  - Encode the problem in some resonable way (like a string $S$)
  - Can create a solution to the problem in time $O(|S|^k)$, for some constant $k$.

FR-248: **NP**

- Nondeterministic Polynomial (NP) problems:
  - Given a solution, that solution Can be verified in polynomial time

- If we could guess a solution to the problem (that's the Non-deterministic part), we could verify the solution quickly (polynomial time)
- All problems in P are also in NP
- Most problems are in NP
-

FR-249: **Class NP-Complete**

- A problem is NP-Complete if:
  - Problem is NP
  - *If* you could solve the problem in polynomial time, then you could solve *all* NP problems in polynomial time
- Reduction:
  - Given problem A, create an instance of problem B (in polynomial time)
  - Solution to problem B gives a solution to problem A
  - If we could solve B, in polynomial time, we could solve A in polynomial time

FR-250: **Proving NP-Completeness**

- Once you have the first NP-complete problem, easy to find more
  - Given an NP-Complete problem $P$
  - Different problem $P'$
  - Polynomial-time reduction from $P$ to $P'$
  - $P'$ must be NP-Complete

FR-251: **Approximation Ratio**

- An algorithm has an *approximation ratio* of $\rho(b)$ if, for any input size $n$, the cost of the solution produced by the algorithm is within a factor of $\rho(n)$ of an optimal solution

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

- For a maximization problem, $0 < C \leq C^*$
- For a minimizaion problem, $0 < C^* \leq C$

FR-252: **Approximation Ratio**

- Some problems have a polynomial solution, with $\rho(n) = c$ for a small constant $c$.
- For other problems, best-known polynomial solutions have an approximation ration that is a function of $n$
  - Bigger problems $\Rightarrow$ worse approximation ratios

FR-253: **Approximation Scheme**

- Some approximation algorithm takes as input both the problem, and a value $\epsilon > 0$
  - For any fixed $\epsilon$, $(1 + \epsilon)$-approximation algorithm
  - $\rho(n) = 1 + \epsilon$
- Running time increases as $\epsilon$ decreases