Comp 352

Time Complexities

① `for(i=0; i<n; i++)` → Execute n+1 times
→ O(1) `print i` → Execute n times
⇒ $T(n) = n * O(1)$ ⇒ $\boxed{O(n)}$

```
for(i=n; i>n; i--)
    print i → Execute n times
```

```
for(i=0; i<n; i=i+2)    *Less executes
    print i → Execute n/2 times
```
$T(n) = \frac{n}{2} O(1) = O(\frac{n}{2}) = \boxed{O(n)}$

② `for(i=0; i<n; i++)` → Execute n+1
`for(j=0; j<n; j++)` → Execute $n \cdot (n+1)$
`print j` → $n \cdot n$
$T(n) = n \cdot n = n^2 \cdot O(1) = \boxed{O(n^2)}$

③
```
for(i=0; i<n; i++)
    for(j=0; j<i; j++)
        print j
```

i	j	#times
0	0	0
1	0	1
2	0, 1	2
3	0, 1, 2	3
⋮	⋮	⋮
n		n

$1+2+3+\cdots+n = \frac{n(n+1)}{2}$

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

$T(n) = \frac{n^2}{2} + \frac{n}{2} = n^2 = O(n^2)$

Less # of times ran, same time complexity

④ `p=0; for(i=1; p<=n; i++)`
`p=p+i;`

i	p
1	0+1=1
2	1+2=3
3	1+2+3
4	1+2+3+4
⋮	
K	1+2+3+4....+K

Assume $p>n$

$\therefore p = \frac{K(K+1)}{2}$

$\frac{K(K+1)}{2} > n \Rightarrow K^2 > n$

$K > \sqrt{n}$

$\therefore \boxed{O(\sqrt{n})}$

⑤ `for(i=1; i<n; i=i*2)`
`printi;`

Assume $i>n$

$i = 2^K$

$2^K >= n$

$\log_2 2^K \geq \log n$

$K \geq \log_2 n$

i
1
$1 \cdot 2 = 2$
2^2
$2^2 \cdot 2 = 2^3$
$2^3 \cdot 2 = 2^4$
⋮
2^K

$\boxed{T(n) = O(\log n)}$

n=8 i	n=10 i
1	1
2	2
4	4
8	8
3	4

$\log 8 = 3$

ceil of log $\rightarrow \lceil \log 10 \rceil = 4$

$\log 10 = 3.2$

16

⑥ `for(i=n; i>=1; i=i/2)`
`printi;`

Assume $i<1$

$\therefore \frac{n}{2^K} = 1$

$n = 2^K$

$K = \log_2 n$

i
n
n/2
n/4
⋮
$\frac{n}{2^K}$

$\boxed{O(\log n)}$

⑦ `for(i=0; i*i<n; i++)` { `i*i<n`, `i*i>=n`, $j^2 >= n$, $i = \sqrt{n}$

`print i`

$O(\sqrt{n})$

⑧ `for(i=0; i<n; i++)` *Not nested

`print i`

`for(j=0; j<n; j++)`

`print i`

$T(n) = O(n) + O(n) = n+n = 2n \Rightarrow O(2n) = \boxed{O(n)}$

⑨ `p=0` `for(i=1; i<n; i=i*2)` $\Rightarrow \log n$ { $p = \log n$

`p++;`

`for(j=1; j<p; j=j*2)` $\Rightarrow \log p$ } $\Rightarrow \log(\log n)$

`print i`

$T(n) = \boxed{O(\log(\log n))}$

⑩ `for(i=0; i<n; i++)` $\Rightarrow n$

`for(j=1; j<n; j=j*2)` $\Rightarrow \log n$ { $T(n) = n \log n$, $T(n) = \boxed{O(n \log n)}$

`print`

Summary

`for(i=0; i<n; i++)` $= n = O(n)$

`for(i=0; i<n; i=i+2)` $= \frac{n}{2} = O(n)$

`for(i=n; i>1; i--)` $= n = O(n)$

`for(i=1; i<n; i=i*2)` $= \log_2 n = O(\log n)$

`for(i=1; i<n; i=i*3)` $= \log_3 n = O(\log_3 n)$

`for(i=n; i>n; i=i/2)` $= \log_2 n = O(\log n)$

Hilroy

Time Complexity of if & while

```
while (condition) { do
    printi  n>=0 }   print          n>0
                     while(condition)
```

① `i=0 while(i<n){` → n+1 times
`printi` → n times
`i++i` → n times
`}`
→ O(n)

*Can re-write while loop into for loop#

② `a=1; while(a<b){`
`printi`
`a=a*2i`
`}`

a
1
1x2
1x2x2
1x2x2
⋮
2^K

$a >= b$
$\therefore a = 2^K$
$2^K = b$
$K = \log_2 b$

$T(n) =$ O(log n)

③ `i=n; while(i>1){`
`printi`
`i=i/2i`
`}`

i
n
$\frac{n}{2}$
$\frac{n}{2^2}$
⋮
$\frac{n}{2^K}$

Assume $\frac{n}{2^K} <= 1$
$n = 2^K$
$\log_2 n$

Can write every while loop into a for loop

$T(n) =$ O(log n)

```
i=1; K=1; while(K<n){
    printi
    K=K+i;
    i++i
}
```

i	K
1	1
2	1+1=2
⋮	⋮
m	$\frac{m(m+1)}{2}$

$\Rightarrow$ O(√n)

```
while(m!=n){
    if(m>n) m=m-n;
    else n=n-m;
}
```

m	n
6	3
3	3

m	n
5	5

min: O(1) ⇒ **O(n)**

```
AlgorithmTest(n)
        if(n<5)
O(1) ⟶      print(n);
        else
O(n) ⟶      for(i=0;i<n;i++)
                print(n)
```

Take if/else that has larger Time complexity

Time Complexity is largest if/else, so T(n) = **O(n)**

Types of Time Functions

O(1) —— Constant T(1) = O(1), T(5) = O(1)

O(log n) —— logarithmetic

O(n) —— linear T(500n + 3500) = O(n)

$O(n^2)$ —— Quadratic

$O(n^3)$ — Cubic

$O(2^n)$ — Exponential

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < 3^n \cdots < n^n$

Asymptotic Notation

O big-oh => Upper Bound

Ω big-omega => Lower Bound

Θ Theta -> Average Bound —— Most useful

Big-Oh: The function $f(n) = O(g(n))$ iff exist positive constants C & n_0 such that $f(n) \leq C * g(n)\ \forall\ n \geq n_0$

e.g: $f(n) = 2n+3$

$2n+3 \leq 10n \quad n \geq 1$

$f(n)\ \ C\ \ g(n)$

$\therefore f(n) = O(n)$

*UpperBound is the time complexity & all others greater. Can be the actual time complexity or anything greater *

omega: The function $f(n) = \Omega(g(n))$ iff exist constants C & n_0 such that $f(n) \geq c * g(n)\ \forall\ n \geq n_0$

e.g: $f(n) = 2n+3$

$2n+3 \geq 1n \quad \forall\ n \geq 1$

$C\ \ g(n)$

$f(n) = \Omega(n)$ -> Most Useful

$f(n) = \Omega(\log n)$

$X\ f(n) = \Omega(n^2)\ X$

*Can be a time complexity of anything lower or the actual time complexity (Average), it is lower bound *

Theta: The function $f(n) = \Theta(g(n))$ iff exists positive Constants c_1, c_2 & n_0 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

e.g: $f(n) = 2n+3$

$1 \cdot n \leq 2n+3 \leq 5n$

$c_1\ g(n)\quad f(n)\quad c_2\ g(n)$

$\therefore f(n) = \Theta(n)$ ✓

$f(n) \neq \Theta(n^2)$

$f(n) \neq \Theta(\log n)$

Ex: $f(n) = 2n^2 + 3n + 4$

$2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$

$2n^2 + 3n + 4 \leq 9n^2 \quad n \geq 1 \} \Rightarrow f(n) = O(n^2)$

$c \quad g(n)$

$2n^2 + 3n + 4 \geq n^2 \Rightarrow \Omega(n^2)$

$1n^2 \leq 2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2 \Rightarrow \Theta(n^2)$

Ex: $f(n) = n^2 \log n + n$

$n^2 \log n \leq n^2 \log n + n \leq 10 n^2 \log n \Rightarrow O(n^2 \log n)$

$\Rightarrow \Omega(n^2 \log n)$

$\Rightarrow \Theta(n^2 \log n)$

Ex: $f(n) = n! = n(n-1)(n-2)\ldots 3 \cdot 2 \cdot 1$

$1 \cdot 1 \cdot 1 \cdot 1 \leq 1 \cdot 2 \cdot 3 \ldots \cdot n \leq n \cdot n \cdot n \ldots \cdot n$

$1 \leq n! \leq n^n \Rightarrow O(n^n)$ *Cannot find Θ average

$\Rightarrow \Omega(1)$ bound*

Ex: $f(n) = \log(n!)$

$\log(1 \cdot 1 \cdot 1 \ldots \cdot 1) \leq \log(1 \cdot 2 \cdot 3 \ldots \cdot n) \leq \log(n \cdot n \cdot n \ldots \cdot n)$

$1 \leq \log(n!) \leq \log(n^n)$

$1 \leq \log(n!) \leq n \log n \Rightarrow O(n \log n)$

$\Rightarrow \Omega(1)$

Properties of Asymptotic Notation

General properties

if $f(n)$ is $O(g(n))$ then $a*f(n)$ is $O(g(n))$
e.g: $f(n) = 2n^2+5 = O(n^2)$
then $7 \cdot f(n) = 7(2n^2+5) = O(n^2)$

Reflexive

if $f(n)$ is given then $f(n)$ is $O(f(n))$
e.g: $f(n) = n^2 \Rightarrow O(n^2)$

Transitive

if $f(n)$ is $O(g(n))$ & $g(n)$ is $O(h(n))$ then $f(n) = O(h(n))$
e.g: $f(n) = n$, $g(n) = n^2$, $h(n) = n^3$
n is $O(n^2)$ & n^2 is $O(n^3)$
then n is $O(n^3)$

Symmetric

if $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$
e.g: $f(n) = n^2$, $g(n) = n^2$, $f(n) = \Theta(n^2)$ $g(n) = \Theta(n^2)$

Transpose Symmetric

if $f(n) = O(g(n))$ then $g(n)$ is $\Omega(f(n))$
e.g: $f(n) = n$, $g(n) = n^2$ then n is $O(n^2)$ & n^2 is $\Omega(n)$

if $f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$
$g(n) \leq f(n) \leq g(n)$ so $f(n) = \Theta(g(n))$

Ex: if $f(n) = O(g(n))$ & $d(n) = O(c(n))$
then $f(n) + g(n) = O(g(n) + c(n))$
eg: $f(n) = n = O(n)$, $g(n) = n^2 = O(n^2)$
$f(n) + d(n) = n + n^2 = O(n^2)$

Ex: if $f(n) = O(g(n))$ & $d(n) = O(c(n))$ then
$f(n) * d(n) = O(g(n) \cdot c(n))$
$n \cdot n^2$

Comparison of Functions

①

$n <$	n^2	$< n^3$
2	$2^2 = 4$	$2^3 = 8$
3	$3^2 = 9$	$3^3 = 27$
4	$4^2 = 16$	$4^3 = 64$

② Apply log on both sides

n^2 n^3
$\log n^2$ $\log n^3$
$2 \log n < 3 \log n$

*Remember: $\log ab = \log a + \log b$, $\log \frac{a}{b} = \log a - \log b$
$\log a^b = b \log a$, $a^{\log_c b} = b^{\log_c a}$, $a^b = n \Leftrightarrow b = \log_a n$ #

Ex: $f(n) = n^2 \log n$, $g(n) = n (\log n)^{10}$

Apply Log

$\log(n^2 \log n)$ $\log(n (\log n)^{10})$
$\log n^2 + \log(\log n)$ $\log n + \log(\log n^{10})$
$2 \log n + \log(\log n) > \log n + 10 \log(\log n)$

Ex: $f(n) = 3n^{\sqrt{n}}$ $g(n) = 2^{\sqrt{n} \log n}$

$3n^{\sqrt{n}}$ $2^{\log n^{\sqrt{n}}}$
$3n^{\sqrt{n}}$ $((n^{\sqrt{n}})^{\log_2 2}$
$3n^{\sqrt{n}} >$ $n^{\sqrt{n}}$

Hilroy

Ex: $f(n) = n^{\log n}$ $g(n) = 2^{\sqrt{n}}$

Apply log

$\log n^{\log n}$ $\quad$ $\log 2^{\sqrt{n}}$

$\log n \cdot \log n$ $\quad$ $\sqrt{n} \log_2 2$

$\log^2 n$ $\quad$ $\sqrt{n}$

Apply Log

$2 \log\log n < \frac{1}{2} \log n$

Ex: $f(n) = 2^{\log n}$ $\quad$ $g(n) = n^{\sqrt{n}}$

$\log 2^{\log n}$ $\quad$ $\sqrt{n} \log n$

$\log n \log_2 2$ $\quad$ $\sqrt{n} \log n$

$\log n < \sqrt{n} \log n$

Ex: $f(n) = 2n < g(n) = 3n$

Ex: $f(n) = 2^n$ $\quad$ $g(n) = 2^{2n}$

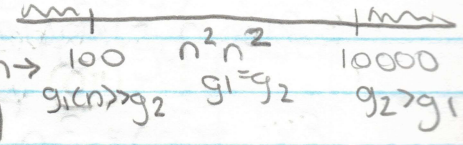
$\log 2^n$ $\quad$ $\log 2^{2n}$

$n \log_2 2$ $\quad$ $2n \log_2 2$

$n < 2n$

Ex: $g_1(n) = \begin{cases} n^3 & n < 100 \\ n^2 & n \geq 100 \end{cases}$

$g_2(n) = \begin{cases} n^2 & n < 10000 \\ n^3 & n \geq 10000 \end{cases}$



Ex:

✓ 1. $(n+k)^m = \Theta(n^m)$ e.g. $= (n+3)^2 = \Theta(n^2)$

✓ 2. $2^{n+1} = O(2^n)$

✗ 3. $2^{2n} = O(2^n)$

✗ 4. $\sqrt{\log n} = O(\log\log n)$

✓ 5. $n^{\log n} = O(2^n)$

Best, Worst & Average Case Analysis

1. Linear Search
2. BinaryTree

Linear Search

Array

0	1	2	3	4	5	6	7	8	9
8	6	12	5	9	7	4	3	16	18

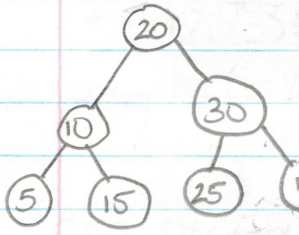
Best Case: Search key element present at first index
Best Case Time: $O(1)$, $B(n) = O(1)$

Worst Case: Search key element present at last index
Worst case Time: $O(n)$, $w(n) = O(1)$

Average Case: All possible case Time / # of cases
Average Time: $1 + 2 + 3 \ldots + n / n = \frac{n(n+1)}{2} \cdot \frac{1}{2} = \frac{n+1}{2}$

$B(n) = 1 = O(1) = \Omega(1) = \Theta(1)$
$w(n) = n = O(n) = \Omega(n) = \Theta(n)$

Binary Search Tree



Best Case: Search elem at root
Best Case Time: $B(n) = 1$
Worst Case: Search elem at leaf
Worst Case Time: $w(n) = \log n$
min $w(n) = \log n$
max $w(n) = n$
* Dependent on height of tree *

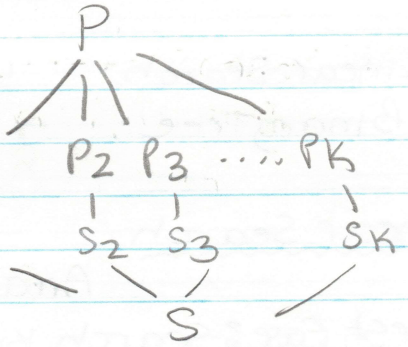
Times

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n! < n^n$

Hilroy

Divide & Conquer

Problem of size n =>
if prob is too big, divide into sub problems, use those solutions to find solution to main problem



Includes: Binary Search, Find MaxMin, Merge, Quick

Recurrence Relations

①

```
T(n) - Void Test(int n)
       {
          if(n>0){
1 ------     print(n);
T(n-1) ---   Test(n-1);
          }
       }
```

$f(n) = n+1 = O(n)$

Test(3) -1
print(3) -1
Test(2)
print(2) -1
Test(1)
print(1)
Test(0) - x

3+1 Calls
n+1 calls
n times

$T(n) = T(n-1)+1 \Rightarrow T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$

$T(n) = T(n-1)+1$
$T(n-1) = T(n-2)+1$
$T(n) = [T(n-2)+1]+1$
$T(n) = T(n-2)+2$
$T(n) = T(n-3)+3$
$\vdots$
$T(n) = T(n-K)+K$

$\rightarrow T(n) = T(n-K)+K$
Assume $n-K=0$ $\therefore n=K$
$T(n) = T(n-n)+n$
$T(n) = T(0)+n$
$T(n) = n+1 \Rightarrow$ Calls
$T(n) = O(n)$

②

```
Void Test(int n){ ——— T(n)
  if(n>0){ ——— 1
    for(i=0; i<n; i++){ — n+1
      print; — n
    }
    Test(n-1) — T(n-1)
  }
```

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

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

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

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

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

$T(n-2) = T(n-3) + n-2$

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

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

$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + \dots + (n-1) + n$

Assume $n-k=0$

$T(n) = T(0) + (n-n+1) + (n-n+2) + \dots + (n-1) + n$

$T(n) = 1 + 1 + 2 + 3 + \dots + (n-1) + n$

$T(n) = 1 + \frac{n(n+1)}{2} \Rightarrow T(n) = O(n^2)$

③

```
Void Test(int n){ — T(n)
  if(n>0){ ⟶ 1
    for(i=1; i<n; i*=2){
      print ⟶ log n
    }
    Test(n-1); ⟶ T(n-1)
  }
```

$T(n) = T(n-1) + \log n$

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+\log n & n>0 \end{cases}$$

$T(n-1) = T(n-2) + \log(n-1)$

$T(n-2) = T(n-3) + \log(n-2)$

$T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n$

Assume $n-k=0$; $T(n) = T(n-k) + \log(n-(k-1)) + \log(n-(k-2)) \dots$

$T(n) = 1 + \log(n!) \Rightarrow T(n) = O(n \log n)$

④ Algorithm Test (int n){
if (n>0){
print(n); → 1
Test(n-1); → T(n-1)
Test(n-1); → T(n-1)
}
}

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

$T(n) = \begin{cases} 1 & n=0 \\ 2T(n-1)+1 & n>0 \end{cases}$

$T(n) = 2T(n-1)+1$
$T(n-1) = 2T(n-2)+1$
$T(n) = 2(2T(n-2)+1)+1$
$T(n) = 2^2 T(n-2)+2+1$
$T(n-2) = 2T(n-3)+1$
$T(n) = 2^2(2T(n-3)+1)+2+1$
$T(n) = 2^3 T(n-3)+2^2+2+1$
$T(n) = 2^k T(n-k)+2^{k-1}+2^{k-2}+\dots+2^2+2+1$

Assume $n-k=0$

$T(n) = 2^n(1)+1+2+2^2+\dots+2^{k-1}$
$= 2^n + 2^n - 1$
$= 2^{n+1}-1 \Rightarrow T(n) = O(2^n)$

Master Theorm For Descreasing Functions

$T(n) = T(n-1)+1 \Rightarrow O(n)$
$T(n) = T(n-1)+n \Rightarrow O(n^2)$
$T(n) = T(n-1)+\log(n) \Rightarrow O(n\log n)$
$T(n) = 2T(n-1)+1 \Rightarrow O(2^n)$
$T(n) = 3T(n-1)+1 \Rightarrow O(3^n)$
$T(n) = 3T(n-1)+n \Rightarrow O(n3^n)$

$T(n) = aT(n-b)+f(n)$, $a>0, b>0$ & $f(n) = O(n^k)$, $k\geq 0$

(1) if $a=1 \Rightarrow O(n\cdot f(n))$
(2) if $a>1 \Rightarrow O(f(n)\cdot a^{n/b})$
(3) if $a<1 \Rightarrow O(f(n))$

Dividing Recurrence Relation

① AlgorithmTest(int n){ — T(n)
  if (n>1){ — 1
    printi — 1
    Test(n/a); — T(n/a)
  }

$T(n) = T(n/a) + 2$
$T(n) = T(n/a) + 1$
$T(n/a) = T(n/2^2) + 1$

$T(n) = T(n/2^2) + 2$
$T(n/2^2) = T(n/2^3) + 1$
$T(n) = T(n/2^3) + 3$, Assume $\frac{n}{2^k} = 1 \Rightarrow n = 2^k$
$k = \log n$

$T(n) = \left(\frac{n}{2^k}\right) + k$
$T(n) = T(1) + \log n \Rightarrow T(n) = 1 + \log n \Rightarrow T(n) = O(\log n)$

② $T(n) = T(n/a) + n$
$T(n/a) = T(n/2^2) + n/a$
$T(n/2^2) = T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$
Assume $\frac{n}{2^k} = 1$, $k = \log n$

$T(n) = T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + \frac{n}{2} + n$

$T(n) = T\left(\frac{n}{2^k}\right) + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} + \dots + \frac{n}{2^2} + n$

$T(n) = T(1) + n\left[\frac{1}{2^{k-1}} + \frac{1}{2^{k-2}} + \dots + \frac{1}{2} + 1\right]$

$T(n) = 1 + n(1+1) \Rightarrow T(n) = 2n+1 \Rightarrow T(n) = O(n)$

③ Void Test(int n){ — T(n)
if(n>1){ — 1
for(i=0; i<n; i++){ — n+1
printf — n
}
Test(n/2); — T(n/2)
Test(n/2); — T(n/2)
}

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

$T(n/2) = T(n/2^2) + \frac{n}{2}$

$T(n/2^2) = T(n/2^3) + \frac{n}{2^2}$

$T(n) = 2(T(\frac{n}{2^2}) + \frac{n}{2}) + n$

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

$T(n) = 2(T(n/2^3) + \frac{n}{2^2}) + 2n$

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

$T(n) = 2^k T(n/2^k) + kn$

$T(n) = n(1) + n\log n$

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

Master Theorm for Dividing Functions

$T(n) = aT(n/b) + f(n)$, $a \geq 1$ $b > 1$ $f(n) = \Theta(n^k \log^p n)$

① $\log_b a$ case 1: if $\log_b a > k$ then $\Theta(n^{\log_b a})$

② k case 2: if $\log_b a = k$ { if $p > -1 = \Theta(n^k \log^{p+1} n)$
$p = -1 = \Theta(n^k \log(\log n))$
$p < -1 = \Theta(n^k)$

case 3: if $\log_b a < k$ { if $p \geq 0 = \Theta(n^k \log^p n)$
if $p < 0 = \Theta(n^k)$

Examples:

① $T(n) = 2T(n/2) + 1$, $a=2, b=2, f(n) = \Theta(1) = \Theta(n^0 \log^0 n)$
$T(n) = \text{Case 1} = \Theta(n^{\log_2 2}) = \Theta(n)$

② $T(n) = 4T(n/2) + n$, $a=4, b=2, f(n) = \Theta(n) = \Theta(n^1 \log^0 n)$
$T(n) = \text{case 1} = \Theta(n^{\log_2 4}) = \Theta(n^2)$

③ $T(n) = 8T(n/2) + n$, $a=8, b=2, f(n) = \Theta(n) = \Theta(n^1 \log^0 n)$
$T(n) = \text{Case 1} = \Theta(n^{\log_2 8}) = \Theta(n^3)$

④ $T(n) = 4T(n/2) + n$, $a=4, b=2, f(n) = \Theta(n) = \Theta(n^1 \log^0 n)$
$T(n) = \text{case 1} = \Theta(n^{\log_2 4}) = \Theta(n^2)$

⑤ $T(n) = 2T(n/2) + n$, $a=2, b=2, f(n) = \Theta(n) = \Theta(n^1 \log^0 n)$
$T(n) = \text{case 2 } (p > -1) = \Theta(n^1 \log^1 n)$

⑥ $T(n) = 4T(n/2) + n^2$, $a=4, b=2, f(n) = \Theta(n^2) = \Theta(n^2 \log^0 n)$
$T(n) = \text{Case 2 } (p > -1) = \Theta(n^2 \log n)$

⑦ $T(n) = 4T(n/2) + n^2 \log^2 n$, $a=4, b=2, f(n) = \Theta(n^2 \log^2 n)$
$T(n) = \text{Case 2 } (p > -1) = \Theta(n^2 \log^3 n)$

⑧ $T(n) = 2T(n/2) + 1 \Rightarrow \Theta(n)$
⑨ $T(n) = 4T(n/2) + 1 \Rightarrow \Theta(n^2)$
⑩ $T(n) = T(n/2) + n \Rightarrow \Theta(n)$
⑪ $T(n) = 2T(n/2) + n^2 \Rightarrow \Theta(n^2)$
⑫ $T(n) = 4T(n/2) + n^3 \log^2 n \Rightarrow \Theta(n^3 \log^2 n)$
⑬ $T(n) = 2T(n/2) + n \log n \Rightarrow \Theta(n \log^2 n)$

Recurrence For Root Function

```
void Test(int n){
  if(n>2){
    printf;
    Test(√n);
  }
}
```

$$T(n) = \begin{cases} 1 & n = 2 \\ T(\sqrt{n}) + 1 & n > 2 \end{cases}$$

$T(n) = T(\sqrt{n}) + 1$

$T(n) = T(\sqrt{n}) + 1$

$T(\sqrt{n}) = T(n^{1/2^2}) + 1$

$T(n^{1/2^2}) = T(n^{1/2^3}) + 1$

$T(n) = T(n^{1/2^3}) + 3 \Rightarrow T(n) = T(n^{1/2^k}) + k$

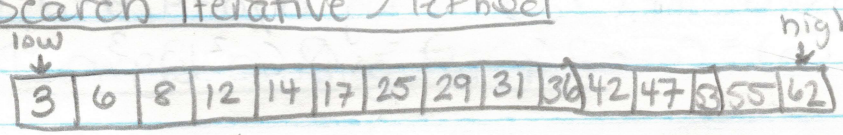
Assume: $n = 2^m$

$T(2^m) = T(2^{\frac{m}{2^k}}) + k$ Assume: $T(2^{m/2^k}) = T(2)$

$m/2^k = 1 \Rightarrow m = 2^k \Rightarrow k = \log m$, $n = 2^m$, $m = \log n$

$k = \log(\log n) \Rightarrow T(n) = \Theta(\log(\log_2 n))$

Binary Search Iterative Method



Must Be sorted, key=42

$mid = \frac{(low+high)}{2}$, if array[mid] > key: high = mid-1

< key: low = mid+1

*Repeat this process until array[mid] == key.

```
BinSearch(A,n,key)
l=0, high=n
while(l≤h){
  mid=(l+h)/2
  if(key==A[mid]) return mid;
  if(key<A[mid]) h=mid-1;
  else l=mid+1;
}
```

min time = $\Omega(1)$

max time = $O(\log n)$

Binary Search Recursive Method

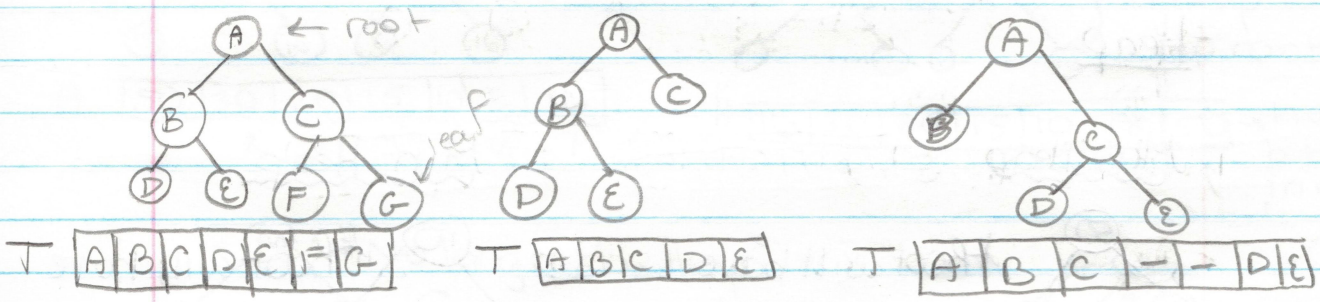
Array=	3	6	8	12	14	17	25	29	31	36	42	47	53	55	62

```
Algorithm(l, h, key){ —— T(n)
    if (l==h){
        if(A[l]==key) return l;
        else return -1;
        }
    else {
        mid=(l+h)/2;
        if(key==A[mid]) return mid; —— 1
        if(key < A[mid]) return Algorithm(l,mid-1,key); —— 1
        else return Algorithm(mid+1, h,key); —— T(n/2)
        }
```

$$T(n) = \begin{cases} 1 & n=1 \\ T(n/2)+1 & n>1 \end{cases}$$

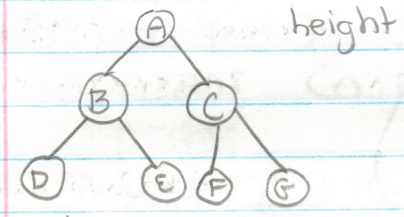
$T(n) = T(n/2)+1 = O(\log n)$

Heap Sort



if a Node is at index - i
its left child is at — 2*i
its right child is at — 2*i+1
its parent is at — $\lfloor \frac{i}{2} \rfloor$

Full Binary Tree / Complete

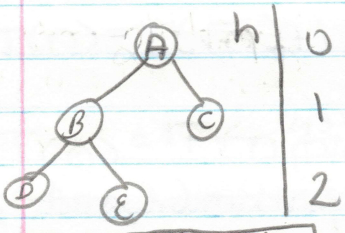


height	
0	
1	
2	

T | A | B | C | D | E | F | G |

* it is full when every layer of tree has full elements *

$2^{h+1} - 1$ Nodes

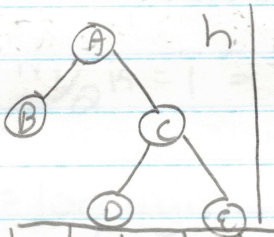


h	
0	
1	
2	

T | A | B | C | D | E |

* Not full binary Tree but is still complete *

* A tree is not complete when missing elements (not filled left to right) *



h	
0	
1	
2	

T | A | B | C | - | - | D | E |

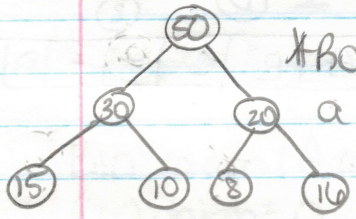
* Not full or complete *

Every full tree is also complete!

Height of Complete Binary Tree is $\log n$

Heap

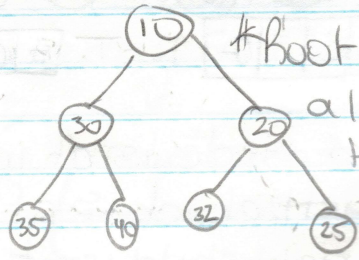
Max Heap



* Root will have a greater value than all its leafs *

T | 50 | 30 | 20 | 15 | 10 | 8 | 16 |

Min Heap

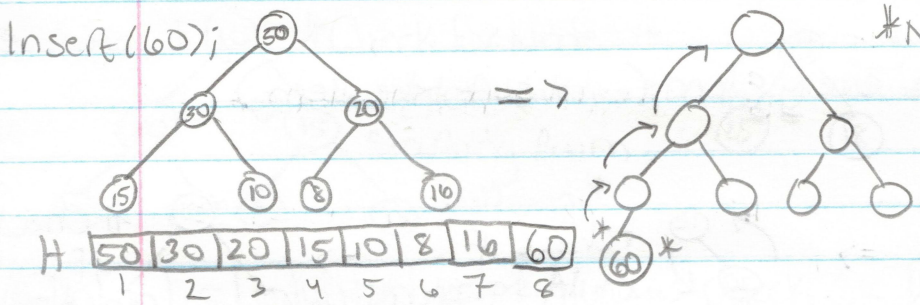


* Root will have a lower value than all its leafs *

T | 10 | 30 | 20 | 35 | 40 | 32 | 25 |

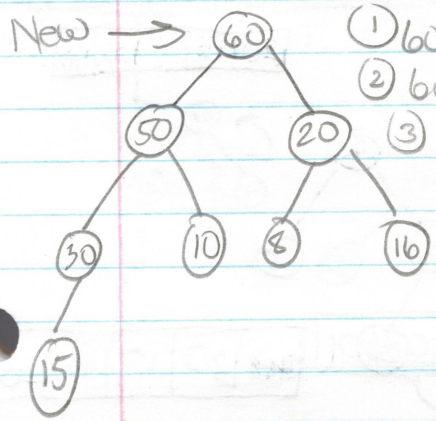
* Heaps Must be Complete Trees *

Insert in Max Heap (down Heap sort like heapify)



Now it is not a max heap Must change it

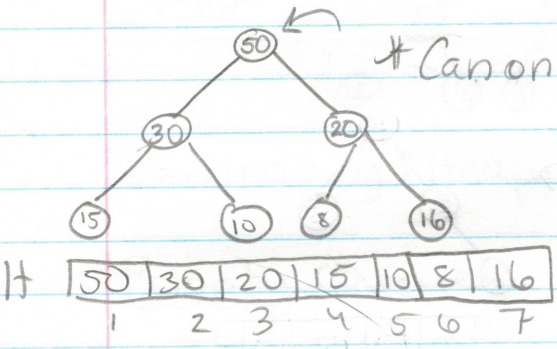
Compare New element with its parents (roots)



(1) 60 was greater then 15, Move to root
(2) 60 was greater then 30, Move to root
(3) 60 was greater then 50, Move to root

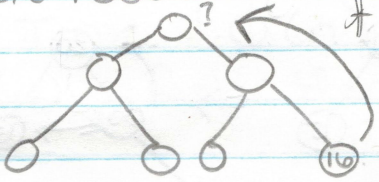
Time for insert is dependent on height
∴ Time = O(logn) upper bound
∴ Time is O(1) to O(logn)

Delete in Max Heap (down heap)



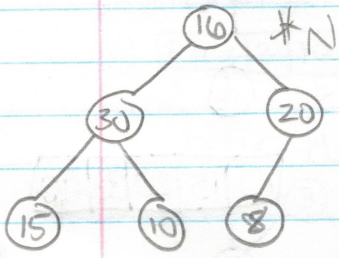
Can only delete root

=>



Put last element in heap to the top root

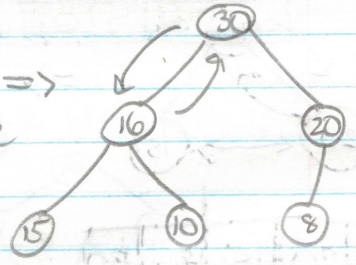
In array but not in heap



No longer Max Heap

Compare two childs, greater one will become top root/parent

=>



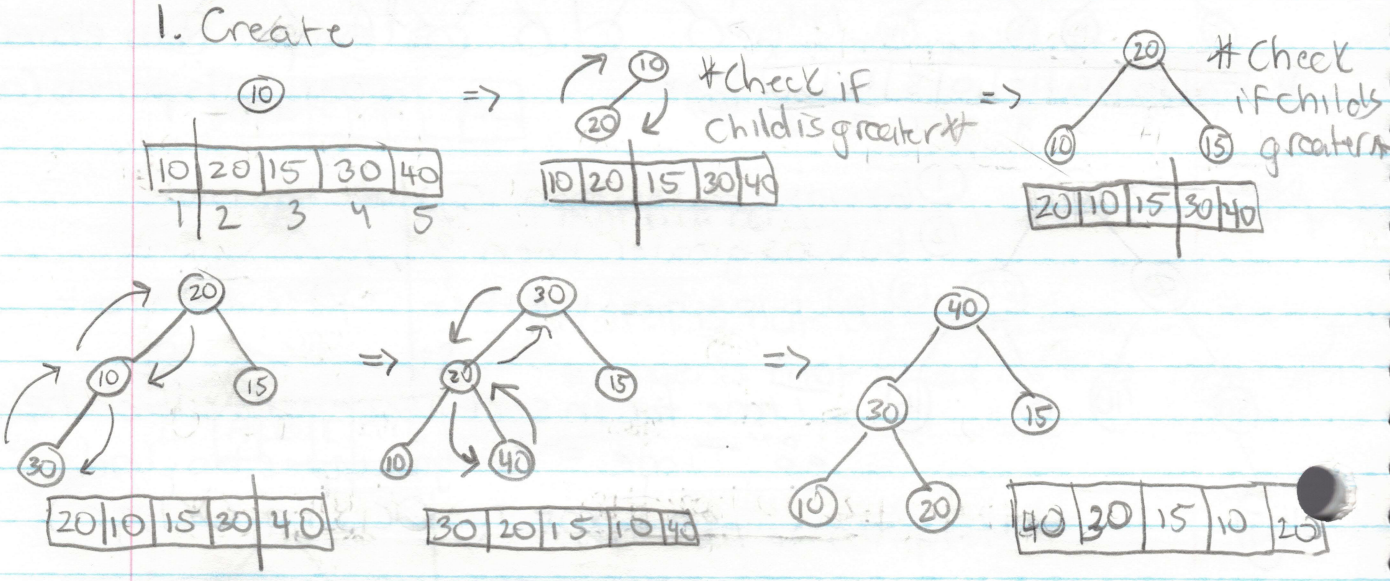
Deletion Time depends on height ∴ O(logn)

Continue checking childs of the last elem to make a max heap (move down if need)

Heap Sorting

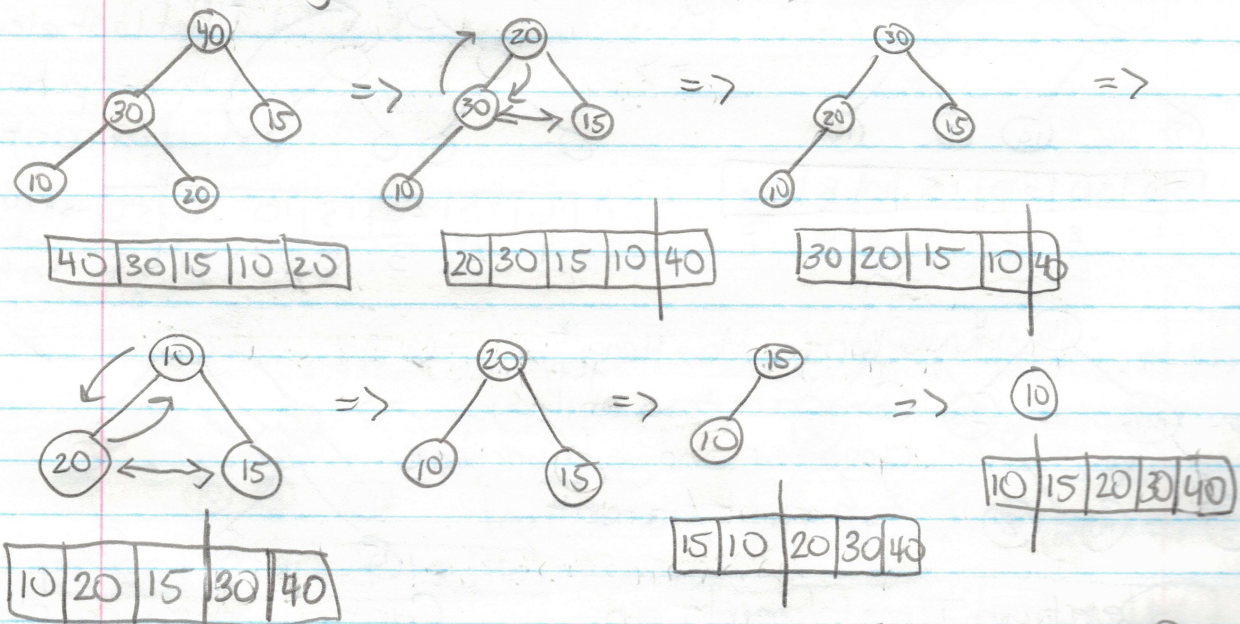
① Create a heap
② Then delete every elem (will sort the elem)

1. Create



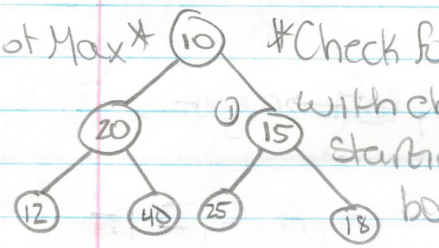
Inserted n elements, takes log n ∴ O(n log n)

2. Deleting

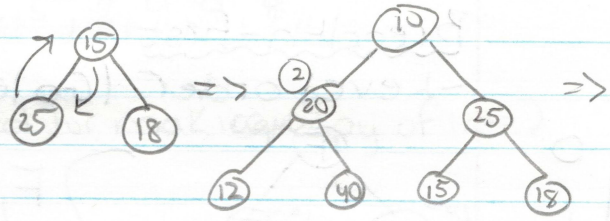


Array: 10 15 20 30 40 Time = O(n log n)

Heapify

Not Max 

Check for Node with children starting from bottom

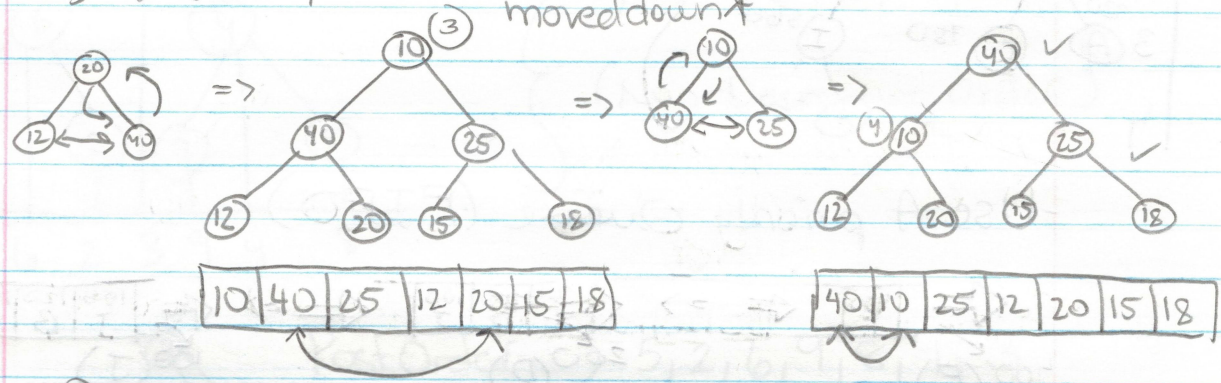


A

1	2	3	4	5	6	7
10	20	15	12	40	25	18

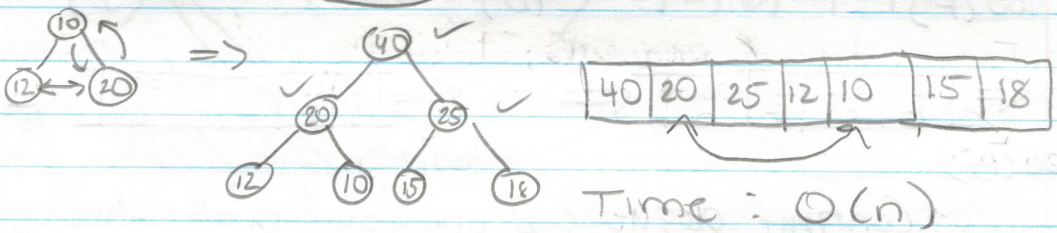
Always check children of node moved down

1	2	3	4	5	6	7
10	20	25	12	40	15	18



1	2	3	4	5	6	7
10	40	25	12	20	15	18

1	2	3	4	5	6	7
40	10	25	12	20	15	18



1	2	3	4	5	6	7
40	20	25	12	10	15	18

Time : O(n)

Summary : Heapify O(n) is faster then create O(nlogn)

Priority Queue

① Small # Priority

A

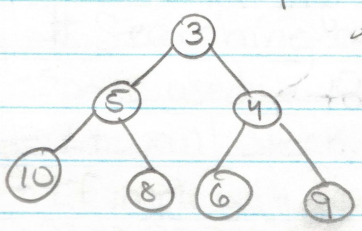
1	2	3	4	5	6	7
8	6	3	10	5	4	9

② Large # Priority

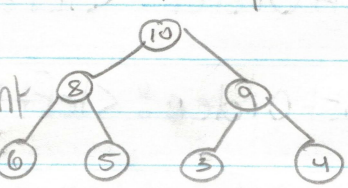
B

8	6	3	10	5	4	9

min Heap



max Heap



heap best DS to implement priority queue

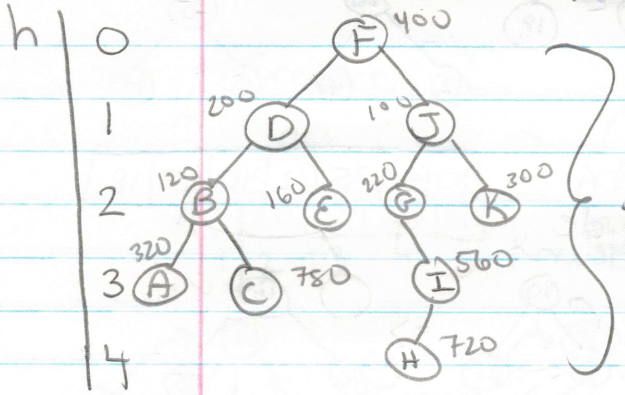
logn insertion

logn deletion

Binary Tree Traversal

Breadth-first

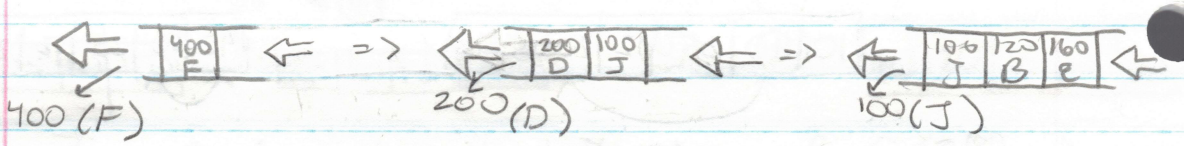
- Level order (Go level by level, left → right)

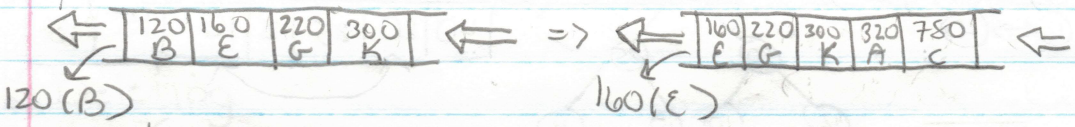


F, D, J, B, E, G, K, A, C, I, H

Moving based on address (Linked List)

- Use A priority Queue (FIFO)





Continue dequeue & enqueue until through tree

Time Complexity = O(n)

Space Complexity = O(1) Best, O(n) Worst/Avg

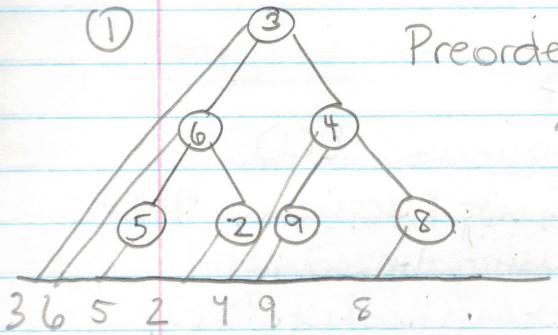
Depth-First

Pre Order: <root><left><right>

In Order: <left><root><right>

Post Order: <left><right><root>

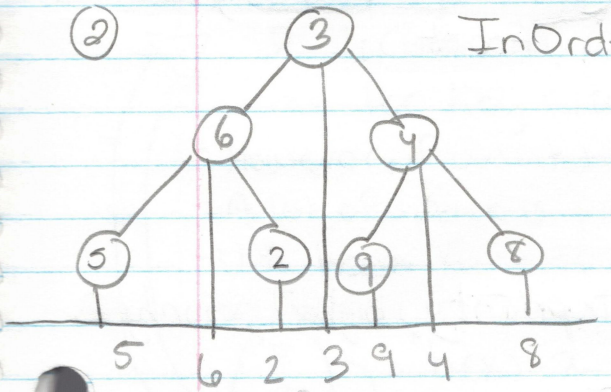
① Preorder



3,6,5,2,4,9,8

(used for making Copy of tree)

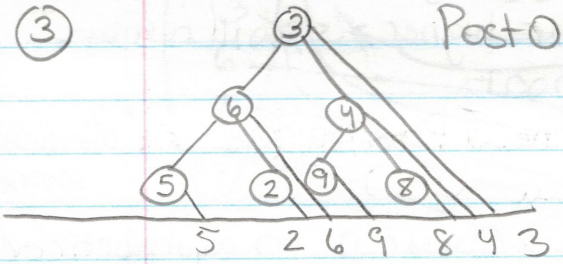
② InOrder



5,6,2,3,9,4,8

(Non-Decreasing Order)

③ PostOrder



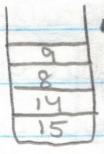
5,2,6,9,8,4,3

(used for deleting Tree

Stacks

* LIFO data structure stores objects in vertical tower *

Methods
- push()
- pop()
- peek()



Top: points to top elem in stack
pop: removes top/return it O(1)
push: Adds elem to top O(1)
peek: return top O(1)

Searching takes O(N)

- Space used is O(n)
- If want stack to not have a limit, use linked list (still O(1) op)
- If imple with growable array (incremental by c strategy push: O(n) Sp↓
  - Same for queue, (Circular Array) enqueue
  - Doubling by 2A Strategy push: O(1) Sp↑

Queues

* (FIFO), first in first out *
- To add to queue use: enqueue, O(1)
- To remove elem from queue, use: dequeue, O(1)
- Must instantiate a Queue using linked list
  Queue<?> queue = new LinkedList<?>();
- To view top element (peek) use: element(), O(1)
- Can use size() & isEmpty(), front();

Priority Queue

(FIFO), first in First out
- stores elements based on their priority, highest priority is first to go
  Queue<?> queue = new PriorityQueue<?>();
- if Queue is higher priority for higher #'s, will remove elements based on their priority
- If you want to reverse order: new PriorityQueue<>(Collections.reverseOrder())
- Methods, enqueue & dequeue O(1)
- if Queue contains strings, Priority Queue is in alphabetical

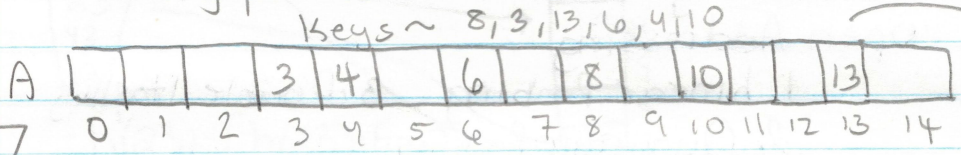
Map

key value
HashMap<?,?> empIds= new HashMap<>();
- Contains no indexes.
- map.put(?,?), enters key & its value, O(1)
- map.get(key), gets value of key, O(1)
- map.remove(key), removes key, value, O(1)

- Uses hash table and handles collision by chaining, adding to head to save time, instead of traversing to end of the linked list after
- Places them by finding hash code of key, then hash(code) % n = index
  (code) = hash code, n = size of array
- Maps override key-value when matching key

Hashing

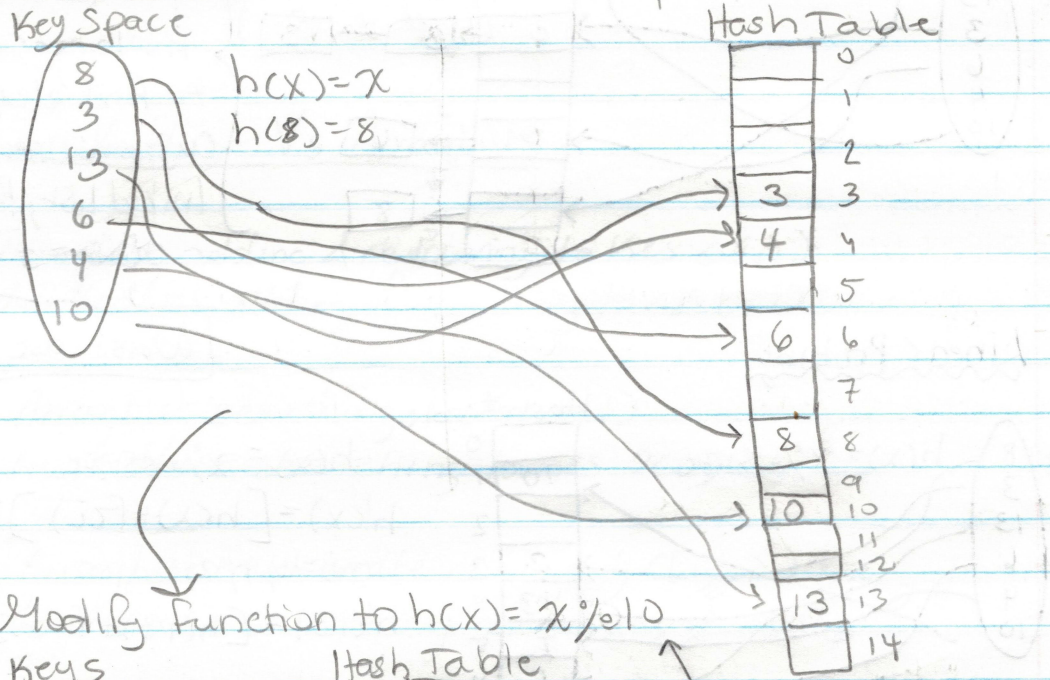
- Linear search, loop through array to find O(n)
- Binary search, (must be sorted) find midpoint, & halve O(logn)
- Hashing provides a search time of O(1)

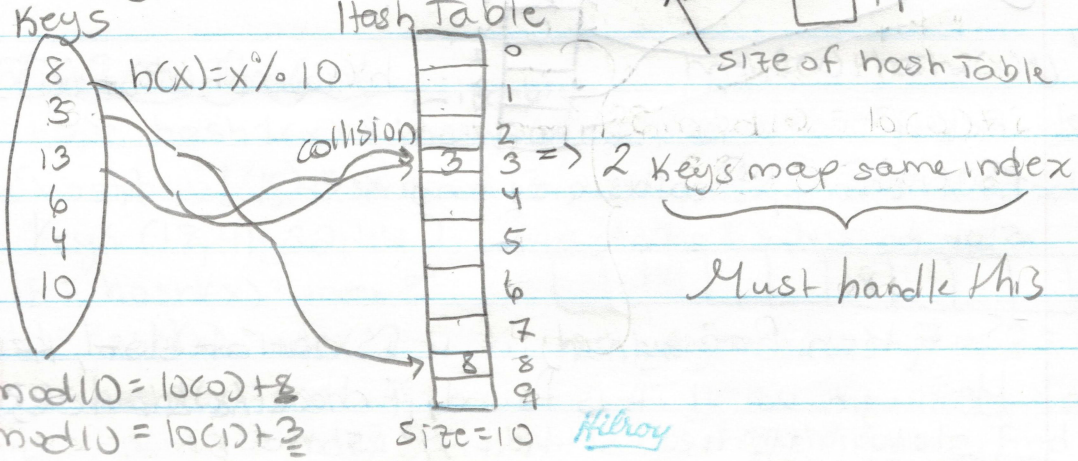


* store value at its index (e.g 8 at 8)*

Search for Key = 12, returns 12 O(1)

- However, if want to store 50, must create more space
- A lot of space is wasted to get a time search of O(1)
- Use hash functions to fix space issue



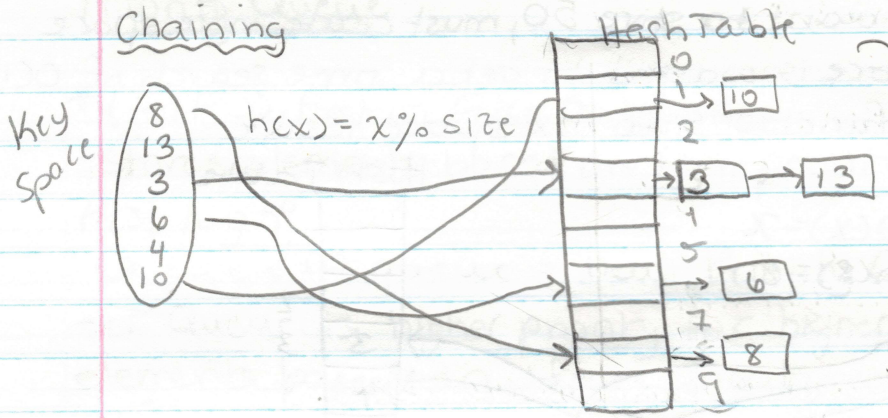


Hilroy

Collision Resolution Methods

1. Open Hashing
   - Chaining - closed addressing
2. Closed Hashing
   - Open Addressing
     1. Linear Probing
     2. Quadratic Probing
     3. Double Hashing

Chaining



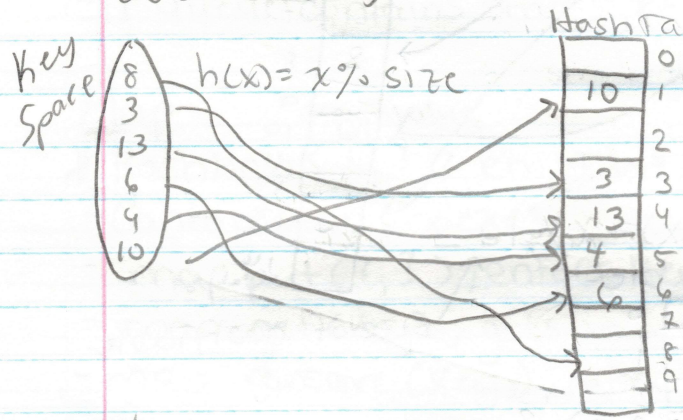
Linked lists at each slot

no longer O(1) to find elem

but now must search linked list, faster than ~~O(n)~~ Table O(n)?

Worst case O(n)

Linear Probing



$h(x) = x \% size$

$h'(x) = [h(x) + f(i)] \% size$

$i = 0, 1, 2, \ldots$

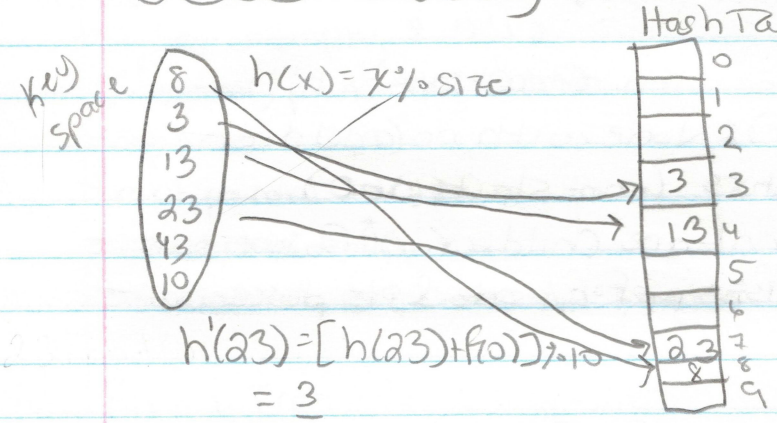
$h'(13) = [h(13) + f(0)] = 3$ ← taken

$h'(13) = [h(13) + f(i)] = 4$ ← not taken

Summary, if collision, find next closest free slot to enter element

- So when finding key, if it is not at slot, keep moving through until it is found, if checking & reaches empty slot that means that # is not present in map

Quadratic Probing



h'(x) = [h(x) + f(i)] % size

f(i) = i²

h'(23) = [h(23) + f(0)] % 10
= 3

h'(23) = [h(23) + f(1)] % 10
= 4 ... → 2²

h'(23) = [h(23) + f(2)] % 10
= 7 (3 + 4)

LinkedList Vs ArrayLists

- ArrayList adding & removing takes O(n)
- LinkedList adding & removing takes O(1)

start at index after collision (start count)

- ArrayList searching takes O(1)
- LinkedList searching takes O(n)

- ArrayList must create new array (bigger) & copy elements over
- LinkedList connect by addresses & nodes

Double Hashing

double collision → hash2(x) gives # you go up by to insert, can loop around if full array

iterable

have hash1(x) & hash2(x)

Ex: hash1(x) = x mod 13, hash2(x) = 7 − x mod 7

Keys (18, 41, 22, 44)

18 = hash1(x) = index 5

41 = hash1(x) = index 2

22 = hash1(x) = index 9

44 = hash1(x) = index 5 ⇒ collision

hash1(x) + j·hash2(x) ← iterable

increments by one when double collision

0	1	2	3	4	5	6	7	8	9	10	11	12
		41			18	44			22	44		

hash1(x) + j·hash2(x) = 7 − 2 = 5 = (5+5) = 10

j = 0 when index is empty

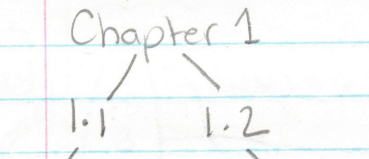
if another collision, find h2(x) that you jump indexes by

Trees

Root: No parent
Internal Node: Node with atleast one child
External Node (leaf): Node with no children
Depth: # of ancestors (root starts at 0)
Siblings: Nodes that are children of same parent
Subtree: Tree consisting of a node & its descendants

Ordered Tree

- Linear ordering defined for children of each node (common)



Tree: can have more than 2ch | at most 2 children
Binary Tree: Can have ~~more children~~
Full B Tree: 0 or 2 children not 1
Complete: 0, 1 or 2 children but no gaps

Methods

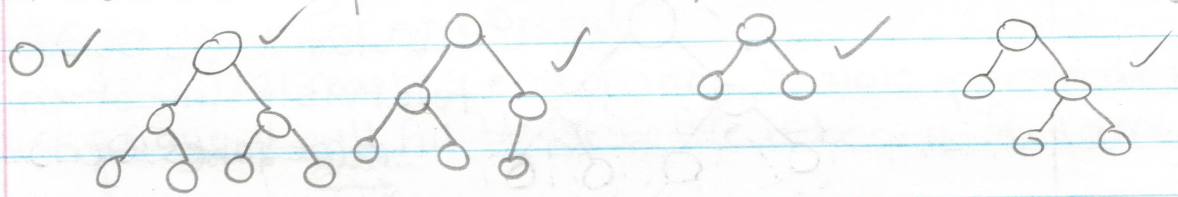
- element(): Return object stored in the position (void)
- size(): Return # of nodes in tree (int)
- isEmpty(): Tests if tree has nodes or not (boolean)
- Iterator(): Return an iterator of all elements stored at nodes of tree
- positions(): Return iterable collection of all nodes of tree
- root(): Return root of tree, error if empty
- parent(p): Return parent of p, error if p is a root
- children(p): Return iterable collection of children of p
  - if p is a leaf, collection is empty
- isInternal(p): Tests whether node p is internal (boolean)
- isExternal(p): Tests whether node p is external (boolean)
- isRoot(p): Test whether node p is root (boolean)
- replace(p, e): Replace elem at p with e, & return element

Performance of Methods

- size, isEmpty : O(1)
- Iterator, positions: O(n)
- replace : O(1)
- root, parent : O(1)
- children: $O(c_v)$
- isInternal, isExternal, isRoot : O(1)

Binary Tree

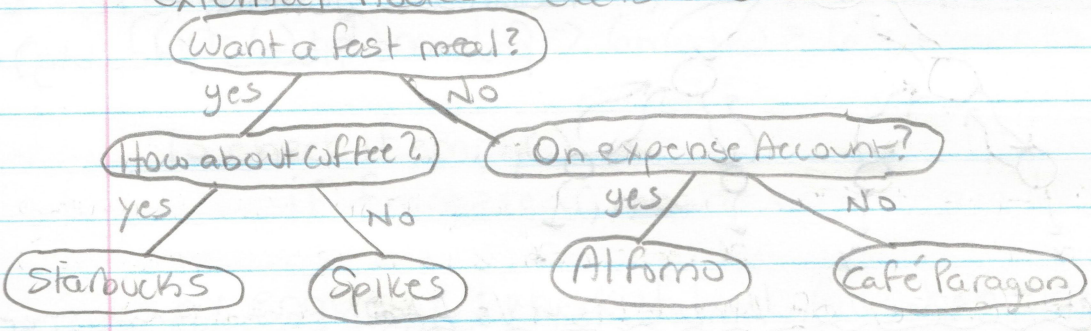
- Internal nodes have at most 2 children (improper if not)
- Node has left & right child (left precedes right)
- A tree consisting of a single node or a tree whose root has an ordered pair of children, each of which is a binary tree



- Good for anthemetic expressions

Decision Tree

- internal nodes : questions with yes/no answer
- external nodes : decisions



Additional Methods

- left(p) : returns left child of p, error if no left child
- right(p) : returns right child of p, error if no right child
- hasLeft & hasRight : returns boolean if child exists of p

Hilroy

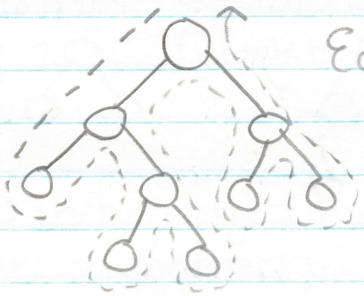
Properties of Binary Trees

Notation:
- n ~ # of nodes
- e ~ # of external nodes
- i ~ # of internal nodes
- h ~ height

Properties:
- $e = i + 1$
- $n = 2e - 1 = 2i + 1$
- $h \leq i$
- $h \leq (n-1)/2$
- $e \leq 2^h$
- $h \geq \log e$
- $h \geq \log(n+1) - 1$

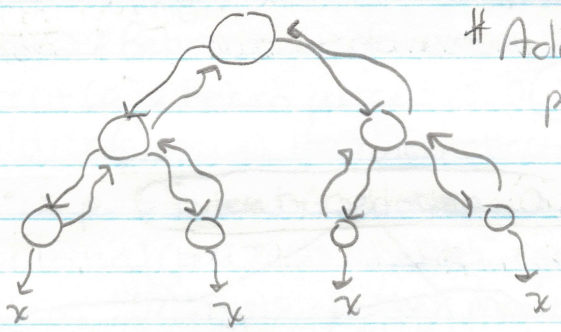
Euler Tour Traversal

- Walks around the tree & visit each node



Each visit takes $O(1)$;
total visit time of n elements
time takes $O(n)$

Linked List Implementation of Binary Tree



Addresses pointing to next & previous nodes (doubly)

Performance of LinkedList Vs Array based BinaryTree

size, isEmpty	$O(1)$	$O(1)$
Iterator, positions	$O(n)$	$O(n)$
replace, root, parent, left, right, sibling, children	$O(1)$	$O(1)$

Array Based Queue

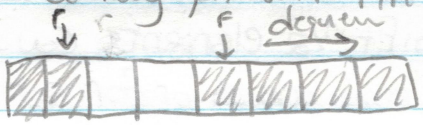
Linked lists have advantage of not having to expand since nodes extend easily

Array

Keep a front & rear pointer
- front points to first element in queue
- rear points to empty slot after last element
- This configuration has space issue, array can have empty slots at the beginning from dequeing, however array will still be full since rear keeps moving to the right

Circular Array

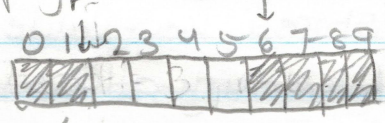
Fixes issue of space issue
- When rear reaches end of array, if slots opened at beginn. of array, it will fill those before it being a full array



Since dequeue removes from front, rear will always be able to fill the empty slots

- To check size()

((N-f)+(r+1)) mod N, ex:



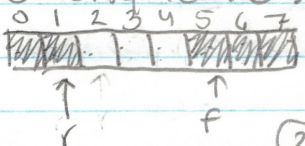
N=10
r=1
f=6

((10-6)+2) mod 10 => 6 mod 10 = 6 elements

- To check if empty()

if front = rear, means no elements between them

- To enqueue: check if full, full when size() = N-1



=> size = ((8-5)+2) mod 8 => 5 mod 8 = 5

if 5 == 8-1 is full && Not full

② Insert elem at r

① Now r = (r+1) mod N => r = 2 mod 8

r = 2

Ordered Maps & Dictionaries

- Entries need to be sorted in map according in total order
- Maintains an order

Ordered Map Methods

- firstEntry(): return entry with smallest key or null if empty
- lastEntry(): return entry with bagest key or null if empty
- floorEntry(K): return entry with largest key or null $\leq K$
- ceilingEntry(K): return entry with smallest key $\geq K$ or null

Performance of Ordered Maps

- get, floorEntry(K) & ceilingEntry(K) takes O(logn) using binary search
- put takes O(n), must shift n/2 elements (worst case)
- remove takes O(n), must shift n/2 elements (worst case)

* Ordered Maps are effective for small maps or for searches *

Dictionary

- Stores key-value pairs (can be of any object types)
- Dichonary allows for same key entries, maps do not (unique)

UnOrdered Dichonary Methods

- get(K): if entry with K return or null, if more K return one of them
- getAll(K): return all entries with key of K (iterable collection)
- put(K,V): insert entry with key K & value V & return this entry
- remove(e): remove entry e from Dict & return it, error if not
- entrySet(): return iterable collection of entries in dichonary
- size() & isEmpty(): Basic

List-Based Dictionary

- Store items of dictionary in a sequence (doubly) arbitrary order
- Performance:
  - put takes O(1) since can add at beginning or end no shifting
  - get & remove takes O(n) since might not be in sequence
  - Better for dictionaries where insert are most common (O(1))

Algorithm getAll(k)
→ Create list L
→ for e: D do if(e.getKey==k)
  L.addLast(e)
  return L

Algorithm put(k,v)
→ Create Entry e = (k,v)
→ D.addLast(e) {D is unordered}
  return e

Algorithm remove(e)
→ B = D.positions()
→ while B.hasNext() do
→ p = B.next
→ if p.element() == e then D.remove(p)
  return e // return null if not found

HashTable Implementation
- If use chainning to handle collisions, in list based operations are O(1)

Search Table Dictionary

- A dischonary implemented by means of a sorted array
- Store items in array sorted by keys, use comparator for keys
- Performance:
  - get takes O(logn) using binary search
  - put takes O(n) since we must shift
  - remove takes O(n) since we must shift
- Only effective for small dictionaries or when searches are most common since search is O(logn) & remove/put is O(n) (slower)

In-order traversal of BST gives sorted

Binary Search Tree does not need to be balanced

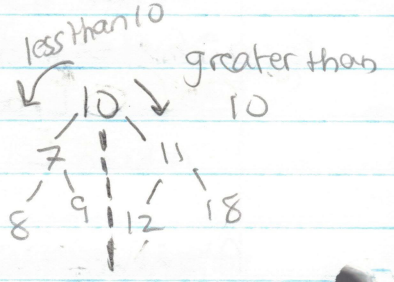
Binary Search Trees

- A binary tree that stores keys (key-value) in its nodes
- Leaf nodes don't store items, they are there to make sure binary tree is in complete/proper form
- Tree's root is midpoint, all #'s to left are less & #'s to right are bigger

Searching

- to search for key k, trace downward path starting at root
- Next node visited depends on comparison of k with node
- if reach a leaf, key was not found

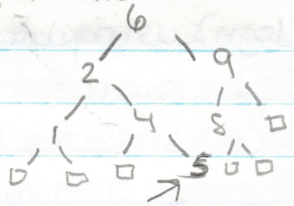
```
Algorithm TreeSearch(k, w)        <- root
    if T.isExternal(w)
        return null
    if k < key(w)
        return TreeSearch(k, T.left(w))
    else if k == key(w)
        return w
    else k > key(w)
        return TreeSearch(k, T.right(w))
```



Binary Tree isn't sorted but #'s less than root are on left & #'s greater on right

Insertion

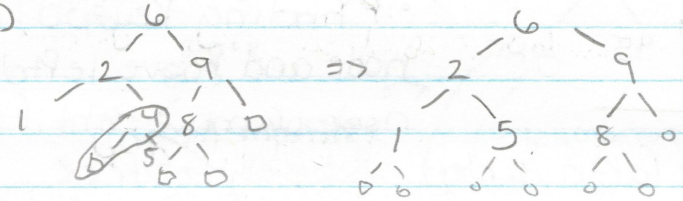
- Assume k is not in tree (must search for it)
- Compare with k & move it all the way down to leaf of internal node Ex: insert(5)



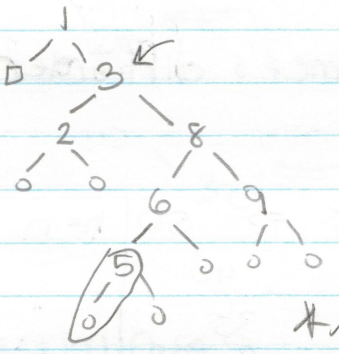
Deletion

- Assume K is in tree (must search for it)
- if node holding K has leaf child, remove node with K & leaf using removeExternal (leaf node) which removes leaf & its parent

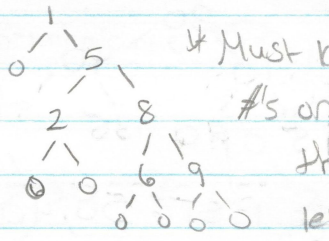
Ex: remove(4)



- if no leaf node Remove(3), find last internal node (keep proper)



=>

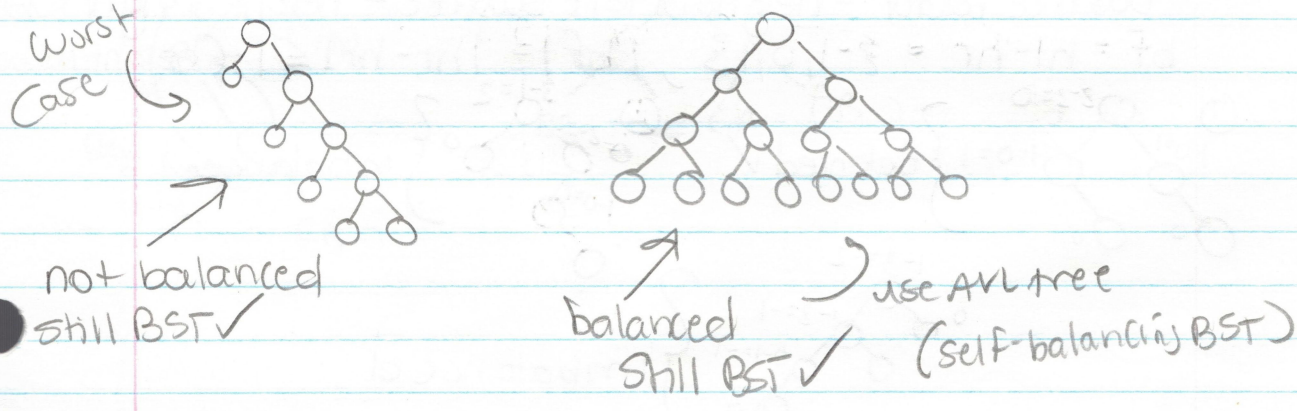


* Must keep proper format
#'s on right are greater than root & #'s on less must be less *
* Doesn't have to be complete *

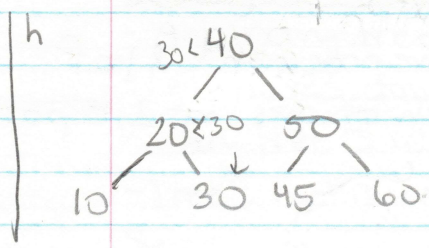
**Binary Tree must have internal nodes with exactly 2 childs **

Performance

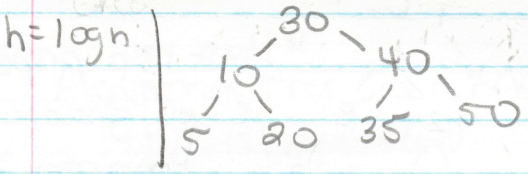
- Consider an ordered map with n items implementing BST of hei h:
  - Space used is O(n)
  - Methods get, floorEntry, ceilingEntry, put & remove take O(h)
  - height h is O(n) in worst case & O(log n) in best

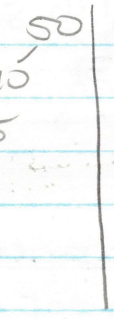


AVL Trees



- In a binary search tree, all #'s to left are smaller than parent node and all #'s to right are greater than parent node
  - if finding key=30, compare with each node and move left if key is less than node or move right if it is greater
- Time taken is dep on height of tree
- Creating BST: 30, 40, 10, 50, 20, 5, 35 *No duplicates*

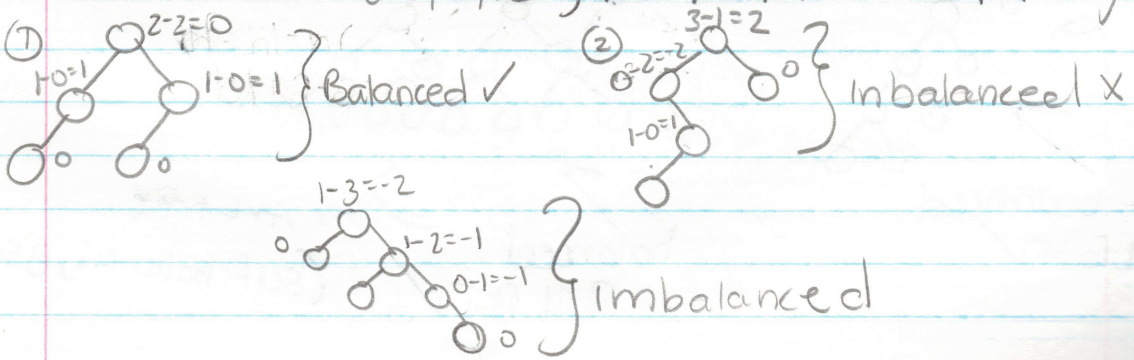


- Create BST: 50, 40, 35, 20, 10, 5 =>  h = n
  downfall is height can either be n or log n based off order of keys

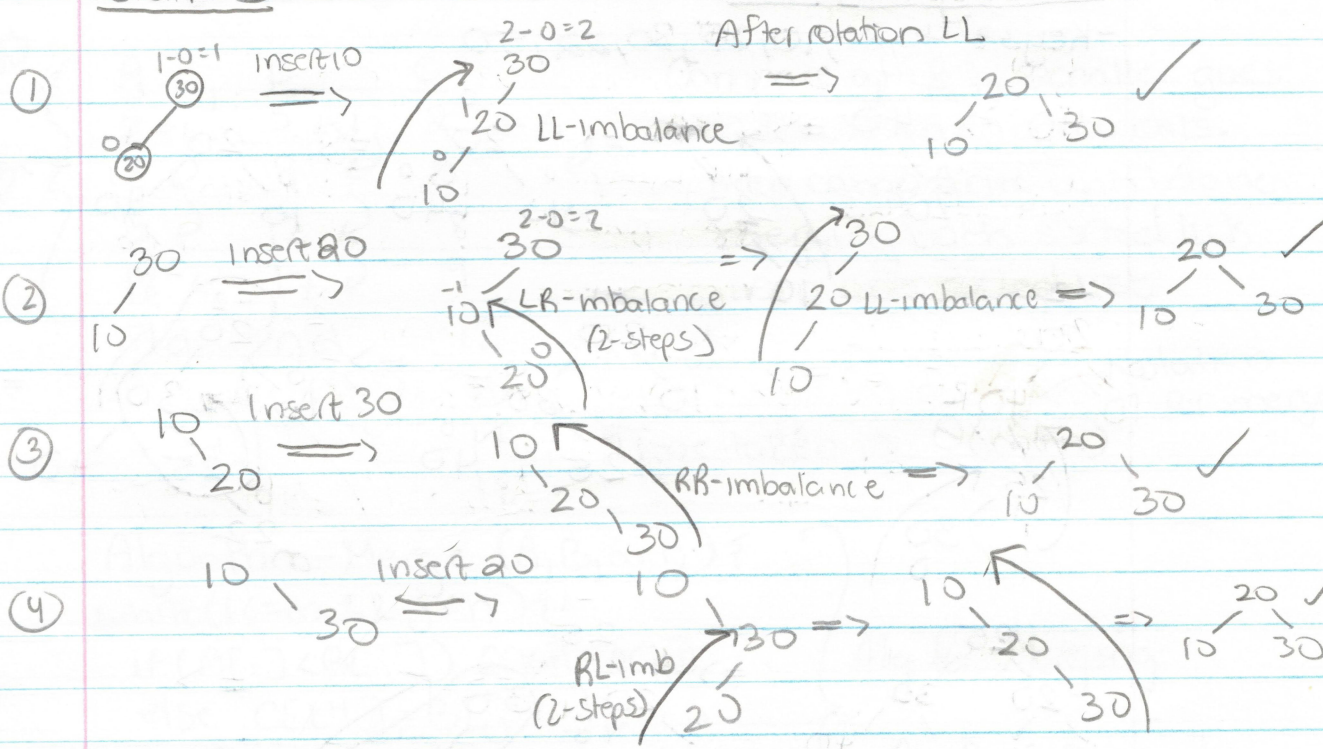
- For keys: 30, 10, 20
  there is 3! tree's to build
  Can be of max or min heigh (objective is to get min height)
- Use AVl tree to rotate 3 nodes (only) to make min tree

AVl

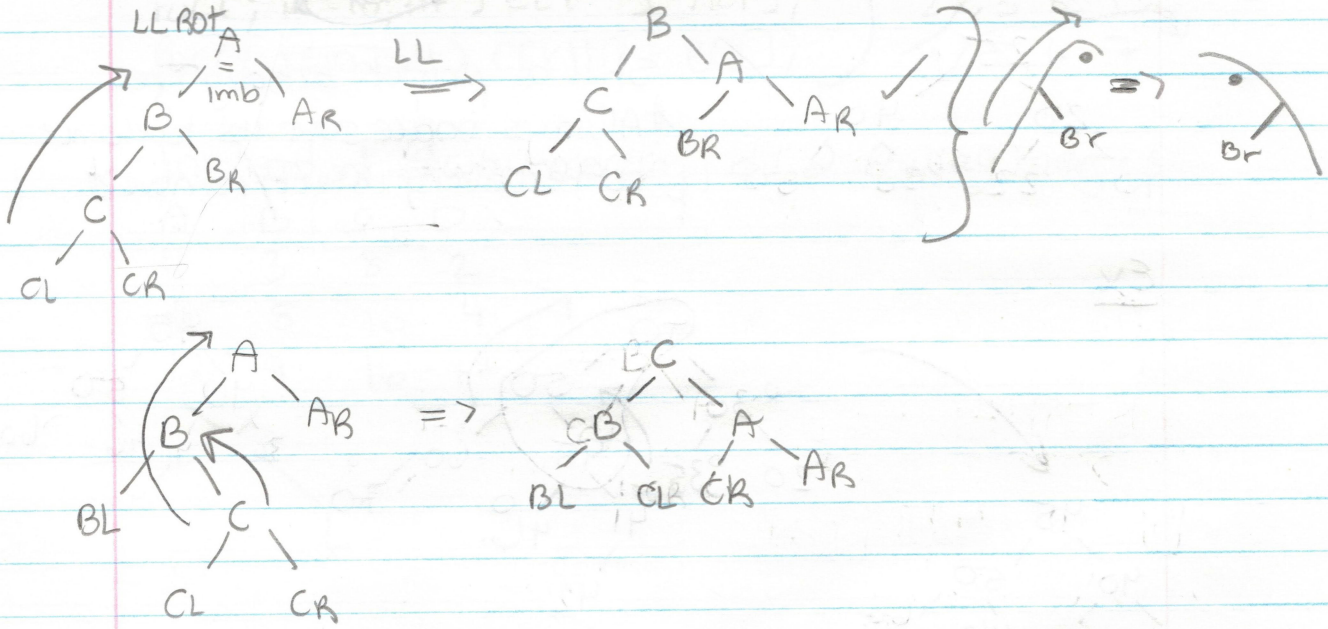
- balance factor = height of left subtree - height of right subtree
  bf = hl - hr = {-1, 0, 1}, |bf| = |hr - hr| ≤ 1 (height not nodes)



rotations

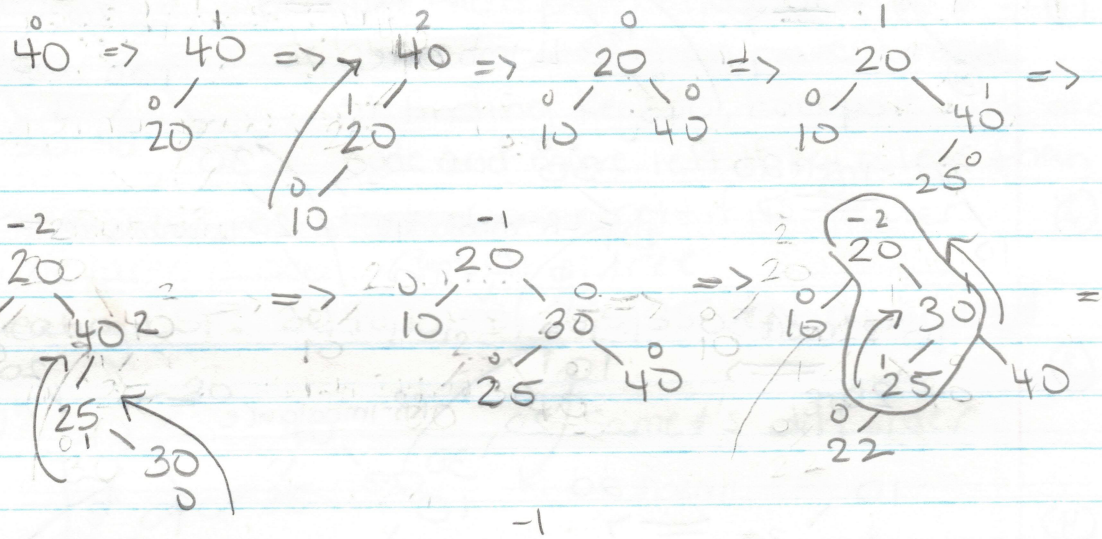


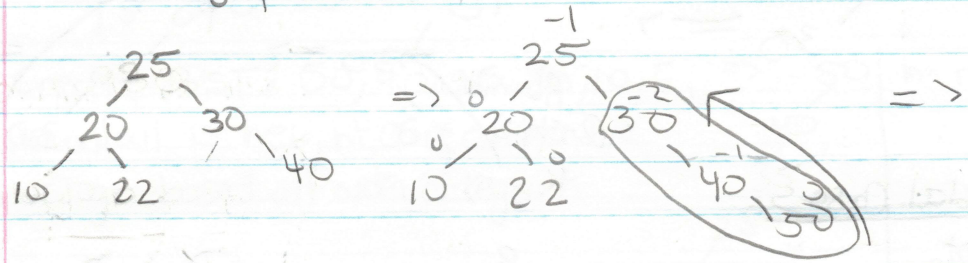
Special Cases

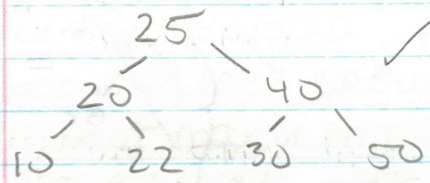


Creating AVL Tree

- Keys: 40, 20, 10, 25, 30, 22, 50



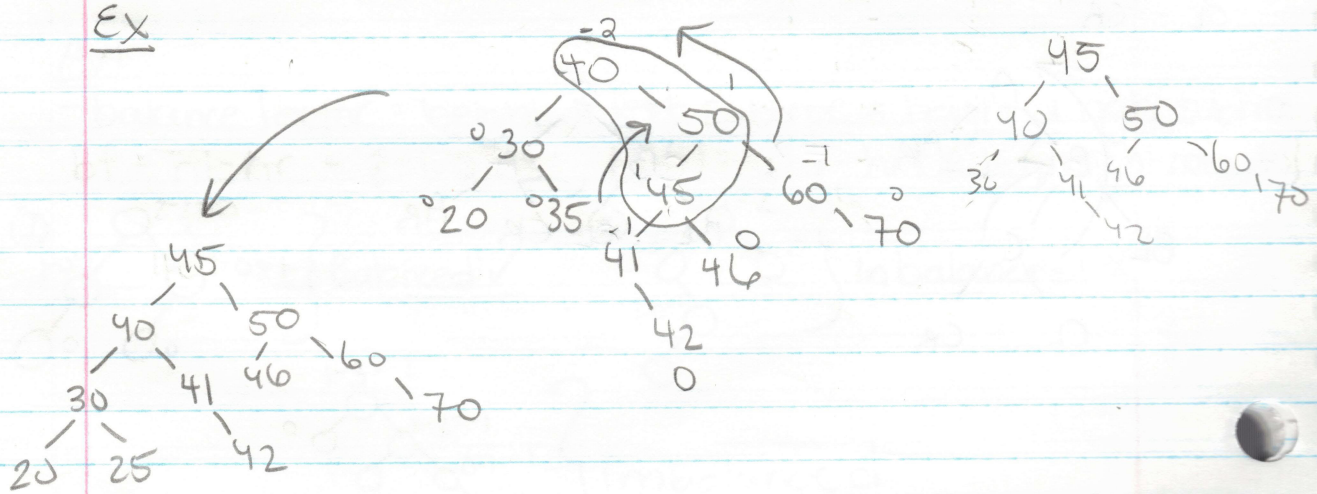




SHIll a proper binary Tree

Always make sure its balance after every insert/remove

Ex



Merge Sorting

Sorted lists A & B

A	B	C
2 ← i	5 ← j	2 ← k
8	9	5
15	12	8
18	17	9
m el	n el	12
		15
		17
		18

Compare at i & j, smaller goes in & that var increments. Keep comparing until done merging both sorted lists into one sorted list

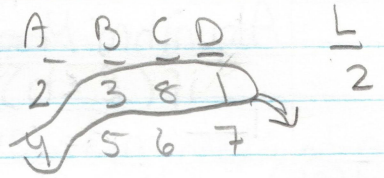
m+n el

Notation for merge

Time taken is $\Theta(m+n)$

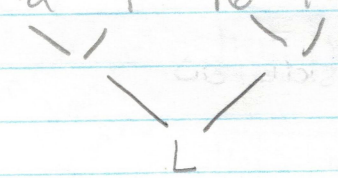
```
Algorithm Merge (A,B,m,n){
while(i<=m && j<=n){
  if(A[i]<B[j]) C[k++]=A[i++];
  else C[k++]=B[j++];
}
for(; i<=m; i++) C[k++]=A[i];
for(; j<=n; j++) C[k++]=B[j];
}
```

Alg for Merging

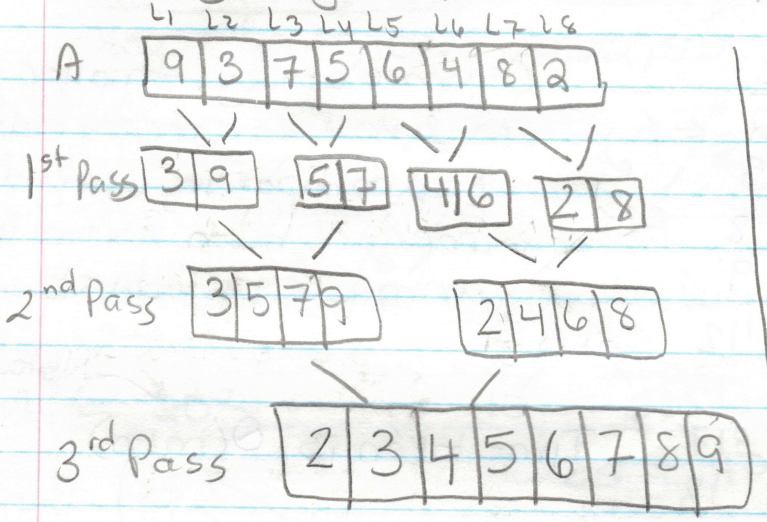


- Instead of 4 way merging, do 2 2-way merges

A	B	C	D
4	3	8	2
6	5	10	4
12	9	16	18



2-Way Merge Sort (Iterative)



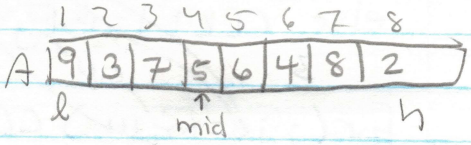
# of passes is log n

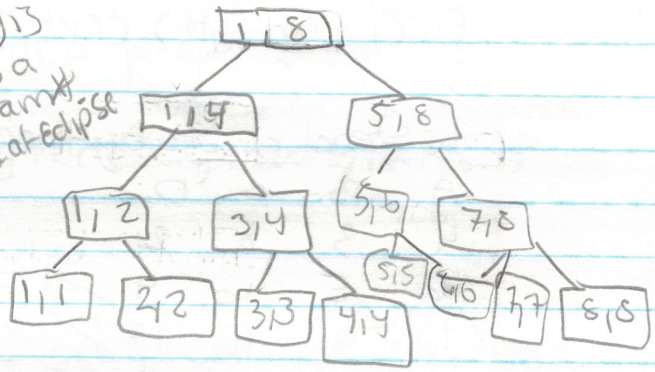
Time = $\Theta(n \log n)$

Merge Sort (Recursive)

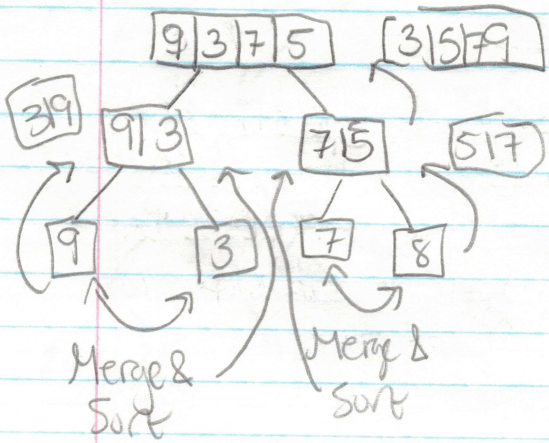
- Follows Divide & conquer
- It is done in pre order traversal

```
Algorithm MergeSort(l, h){
    if(l < h){
        mid = (l+h)/2;          * Array is also a param** Look at clipse
T(n/2)  MergeS(l, mid);
T(n/2)  MergeS(mid+1, h);
n       Merge(l, mid, h);
```





Time take $\Theta(n \log n)$



⇒ other side too

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

Master Theorm

$\Theta(n \log n)$

↑ always nlogn Ω, Θ, O

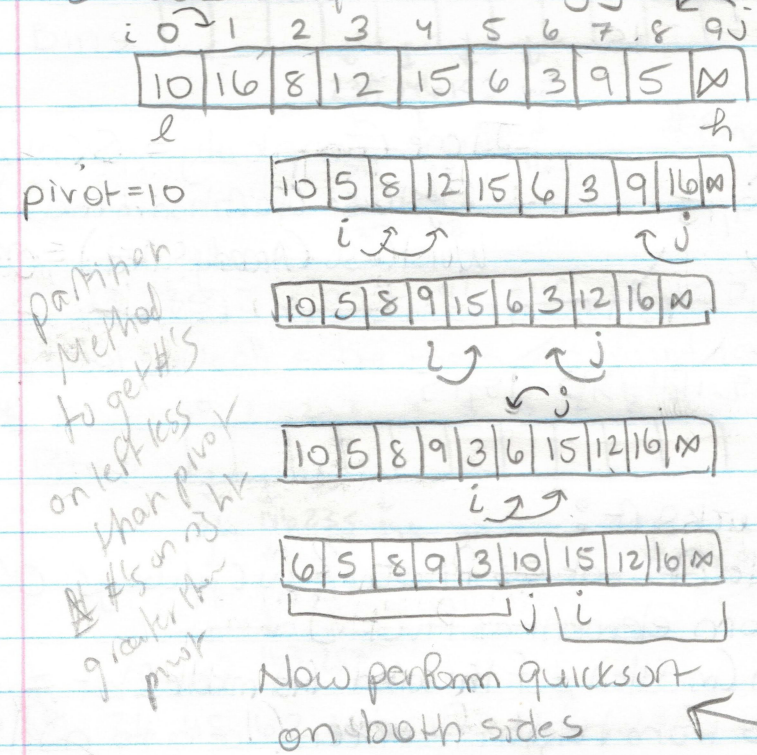
Pros & Cons of Merge Sort

Pros:
1. Large size list*
2. Linked lists (can merge by just changing links)
3. External Sorting (can take big files & sort them in chunks & export, allowing for breaking up of big files & sorting)*
4. Stable (if duplicates, the first that appeared will be sorted first)

Cons:
1. Extra space (not inplace sort), merge into new array
2. No small problem (slower at small lists in comparison)
3. Recursive logn stack space & O(n) space

QuickSort

-Divide & conquer Strategy



partition method to get #'s on left less than pivot & #'s on right greater than pivot

Start at index i at 0 & j at last index. Pivot will be # at index 0. Increm i until # is bigger than pivot #, once found decrem j until reaches # less than pivot #, once found switch #'s at i & j. Repeat this until i passes j. Once happens swap pivot # with # at j. Then

Now perform quicksort on both sides

```
Algorithm Partition(l,h){
   pivot = A[l]; i=l; j=h;
while(i<j){
  do {
   i ++
  }while(A[i]≤pivot)
  do{
   j--
  }while(A[j]>pivot)
  if(i<j) swap(A[i],A[j])
  }
 Swap(A[l], A[j]);
 return j;
}
```

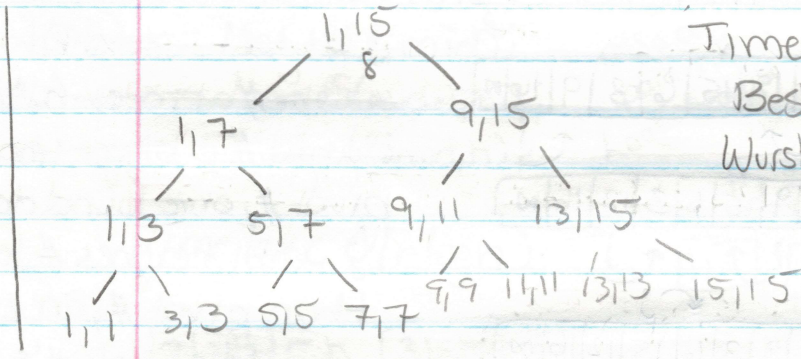
```
Algorithm QuickSort(l,h){
 if(l<h){
   j=Partition(l,h);
 QuickSort(l,j);
 QuickSort(j+1, h);
   }
 }
```

#Hoath At eclipse#

QuickSort Analysis

1 - - - - - - - - - - - - - - 15

7,8,9



Time Complexity = $O(n \log n)$
BestCase (Partition in mid) = $O(n \log n)$
WurstCase (Already sorted) = $O(n^2)$

-To improve QuickSort:
- Select Middle Element as Pivot } w c always $O(n^2)$
- Select Random element as Pivot }
- Select median (middle) # of the sorted first middle & last #, & place that at beginning

Stack size is from height of a tree (log n to n)

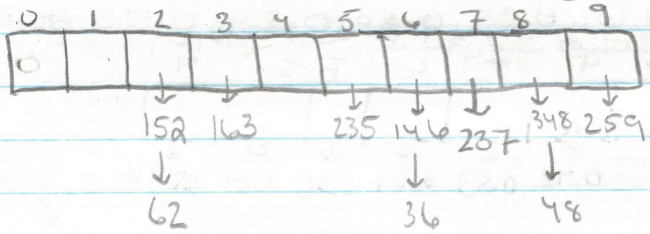
#Look into best ways to chose pivot to be most effic quickort runtime $O(n \log n)$ *

Radix Sort

	0	1	2	3	4	5	6	7	8	9
A	237	146	259	348	152	163	235	48	36	62

Bins: 0 1 2 3 4 5 6 7 8 9

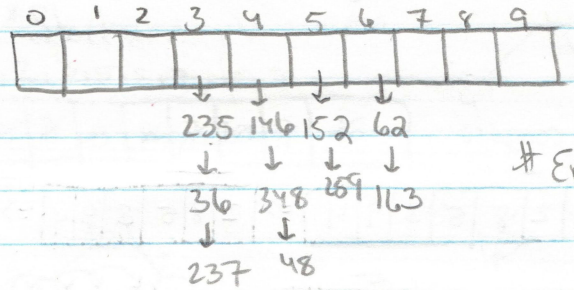
To enter into a bin, check last digit of the #



Now empty stack in FIFO starting from index=0

① 152, 62, 163, 235, 146, 36, 237, 348, 48, 259

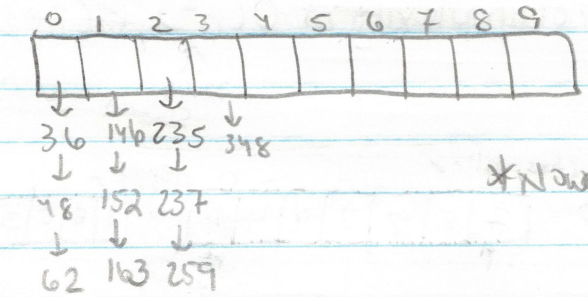
*Not sorted, so now sort them on second last digit $(A[i]/10)\%10$ #



#Empty Bins in FIFO again*

② 235, 36, 237, 146, 348, 48, 152, 259, 62, 163

*Not sorted, sort them on third last digit $(A[i]/100)\%10$



Now empty Bins in FIFO again

③ 36, 48, 62, 146, 152, 163, 235, 237, 259, 348 Sorted

Time dep on largest number, takes O(n), space O(n)

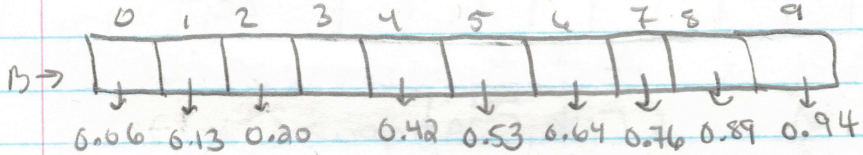
Worst $O(n^2)$

Bucket Sort

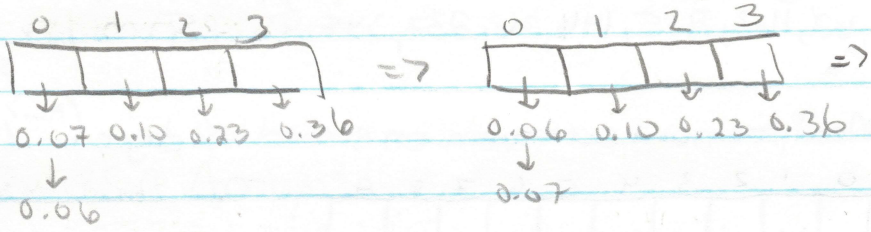
(when all dem placed in same bucket (long chaining))

- worst case $O(n^2)$, BEST/AVERAGE $O(n)$
- Works for floating point #'s 0.0 to 1.0
- Inputs should be uniformly & indep distrib across [0,] to get a runtime of $O(n)$

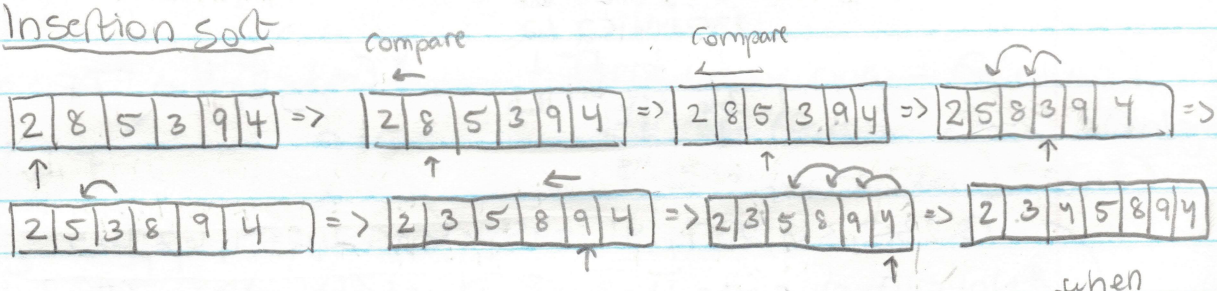
A → 0.79, 0.13, 0.64, 0.39, 0.20, 0.89, 0.53, 0.42, 0.06, 0.94



When collisions, sort collisions using insertion sort



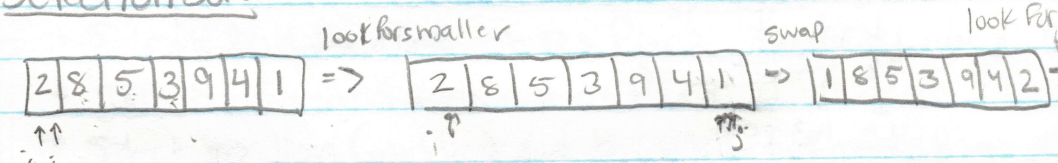
Insertion Sort



Look At algo # Time Complexity of $O(n^2)$, Best $(O(n))$ ← when sorted

↑ reverse sorted

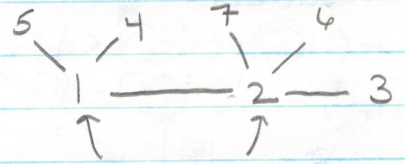
Selection Sort



Best Case/Worst Case/Average $O(n^2)$

Start at beginning, place # in min box, move through whole list replacing current min with lowest min, when finished, swap the index's, then repeat this process starting at second index (keep incrementing by until finish the array.

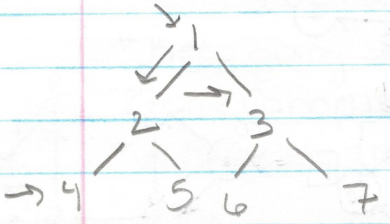
Graphs BFS & DFS



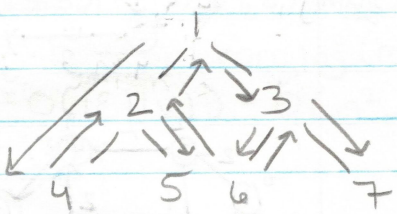
1. Visiting a Vertex
2. Exploration of vertex

BFS: 1, 2, 4, 5, 7, 3, 6 → Explore Vertex by vertex

DFS: 1, 2, 3, 6, 7, 4, 5 → Explore to last vertex, then explore back (can't explore anymore)



BFS: 1, 2, 3, 4, 5, 6, 7 (level order)

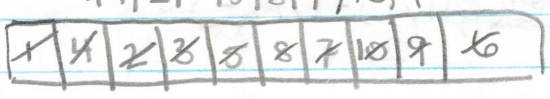


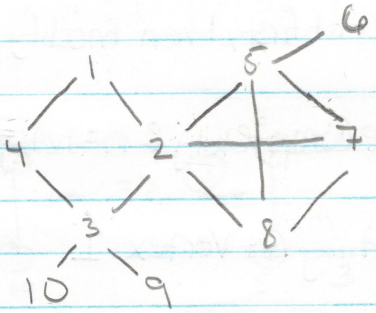
DFS: 1, 2, 4, 5, 3, 6, 7
(Pre order traversal)

BFS: *Start at any Vertex*

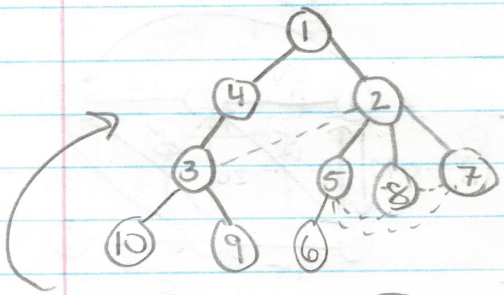
1, 4, 2, 3, 5, 8, 7, 10, 9

Queue



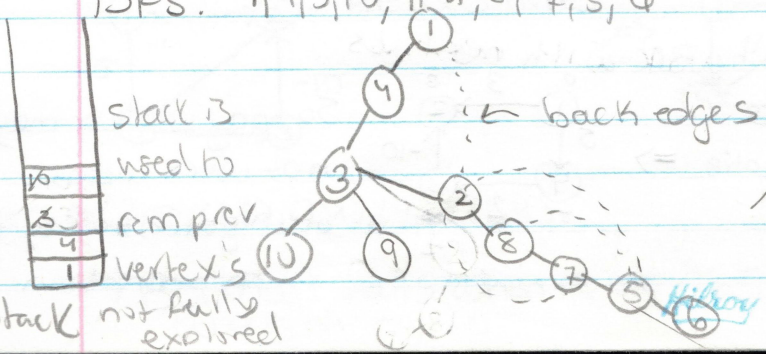


* More than one valid BFS, order does not matter *



BFS spanning Tree

DFS: 1, 4, 3, 10, 9, 2, 8, 7, 5, 6



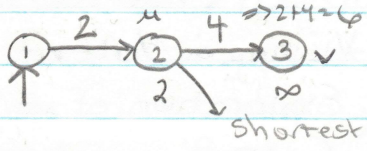
* Must explore vertexs until vertex completely explored then go back & finish exploring *

Finds shortest path from start to all other nodes

* Must visit all nodes, & once visited that node never re-visit *

Non-Arrow graph (Nodes can go to any), Directed (must follow arrows)

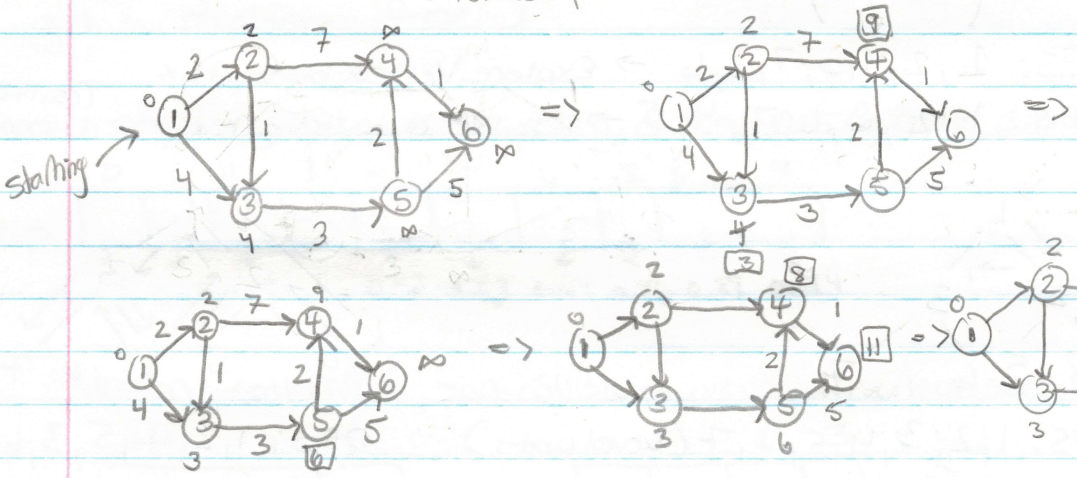
Dijkstra Algorithm



Relaxation:

if(d[u]+c(u,v) < d[v])

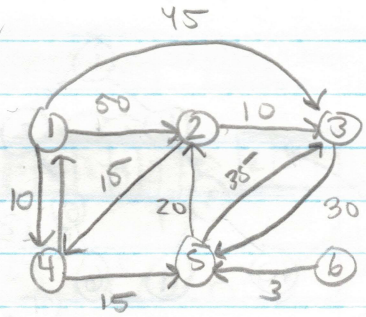
d[v] = d[u] + c(u,v)



* Start at first, then modify, then continue at smallest modified node *

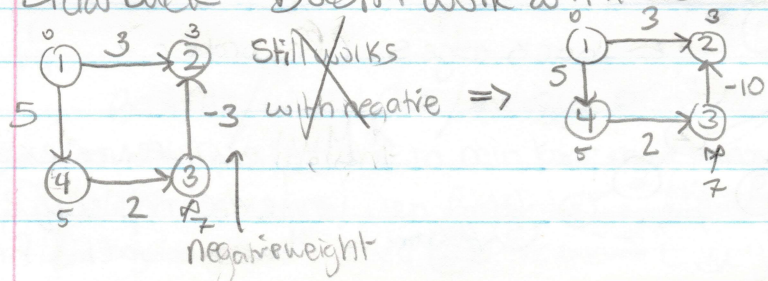
Time Complexity: n = |V| => O(|V||V|) = O(n²)

- Starting At Vertex 1



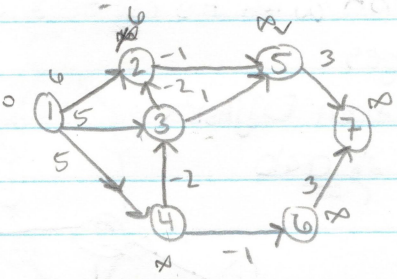
Selected Vertex	2	3	4	5	6
4	50	45	(10)	∞	∞
5	50	45	~~10~~	(25)	∞
2	(45)	45	~~10~~	~~25~~	∞
3	~~48~~	(45)	~~10~~	~~25~~	∞
6	~~48~~	~~48~~	~~10~~	~~25~~	(∞)

- Drawback - Doesn't work with negatives



So may work or may not work incase of negative edges, depends on path direction

Bellman-Ford Alg



$|V| = n = 7$

$|V|-1$ edges

Relaxation

if $(d[u] + C(u,v) < d[v])$

$d[v] = d[u] + C(u,v)$

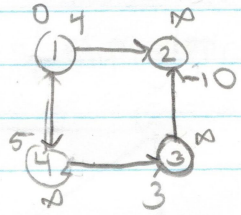
Edgelist → (1,2), (1,3), (1,4), (2,5), (3,2), (3,5), (4,3), (4,6), (5,7), (6,7)

* Keep going edge to edge until weights stop changing (relaxation)

Time Complexity = $\Theta(|E||V|-1) = O(\hat{|E|}\hat{|V|}) = O(n^2)$

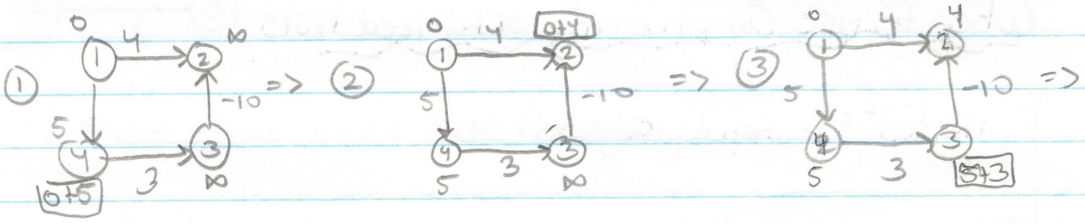
- In Complete graph, max time is $O(n^3)$

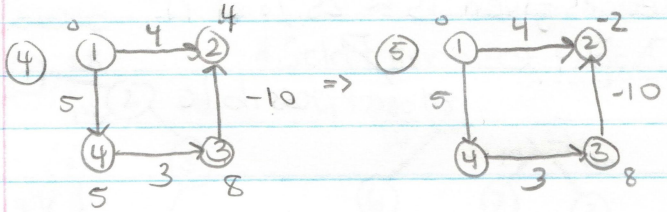
Ex:



edges → (3,2), (1,2), (1,4), (4,3)

relax $n = 4-1 = 3$ times

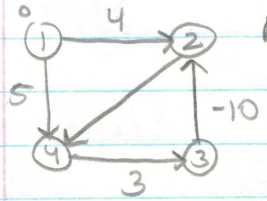




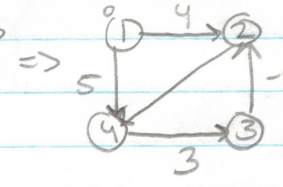
→ Done, Full Relax through all edges 3 times

* So shortest path is at ② with -2

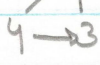
Drawback



Relax 3 times



Issue is a cycle with a weigh of -2, which will always keep changing every relaxation

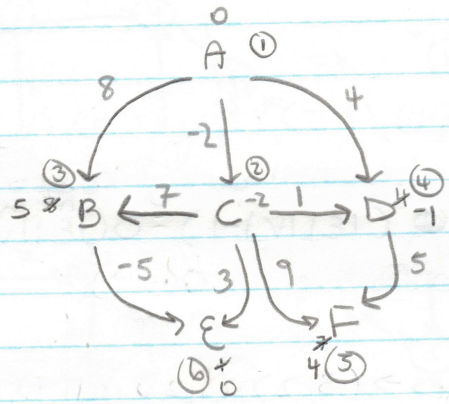


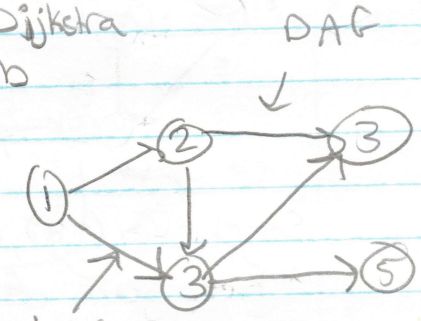
Edges → (3,2), (1,2), (1,4), (4,3), (2,4)

* Bellman Fails when contains negative weight cycle

DAG-based Algo

- Directed acyclic Graph (Graph with no cycles & directed)
- Works with negative-weight edges
- Time complexity of O(n) faster than Dijkstra
- Use topological order => $a \to b \therefore a < b$





Watch Video As Review

When to use Graph array or linked lists

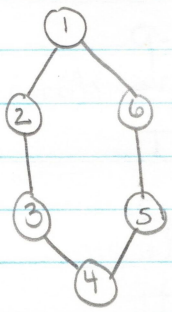
$nodes^2$ = possible edges

Use array when #edges given is $\approx 25\%$ of n^2

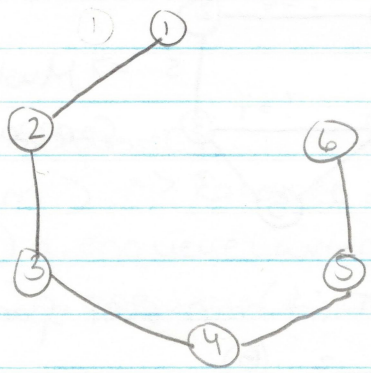
↑ closest possible

Can't Make Cycles

Minimum Spanning Trees (MST) 3.5



$\Rightarrow$



$S \subseteq G$

$S = (V', E')$

$V' = V$

$|E'| = |V| - 1$

$G = (V, E)$

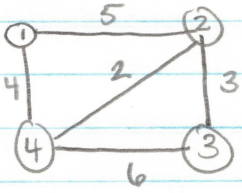
$V = \{1,2,3,4,5,6\}$

$E = \{(1,2),(2,3),(3,4)\ldots\}$

$|V| = n = 6$

$n - 1 = 5$

# of spanning Trees is $|E| C_{|V-1|}$ - # of cycles

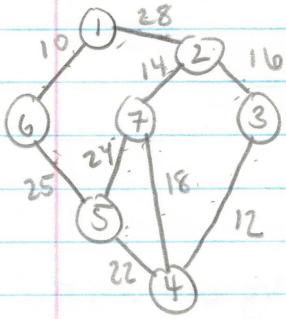


weighted Graph

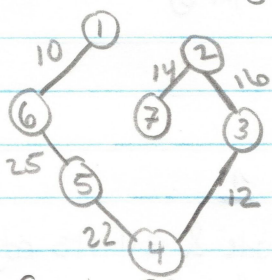
Weighted graphs have min & max cost trees

To Find these trees, use Prims Algo or Kruskal's algo

① Prims



Select smallest Edge

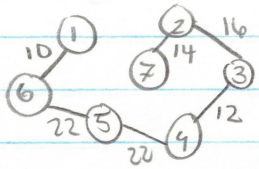


Cost = 99

Must select smallest Edge, & only add connecting smallest edges

Cannot find Spanning trees for disconnected graphs

② Kruscals



Cost = 99

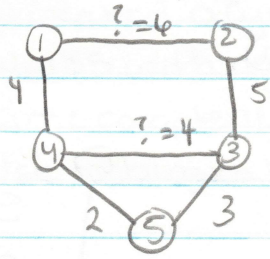
Must pick smallest edges that don't complete a cycle

if min heap is used $\theta(n \log n)$ makes it faster

$O(|V||E|) = \theta(ne) = \theta(n^2)$

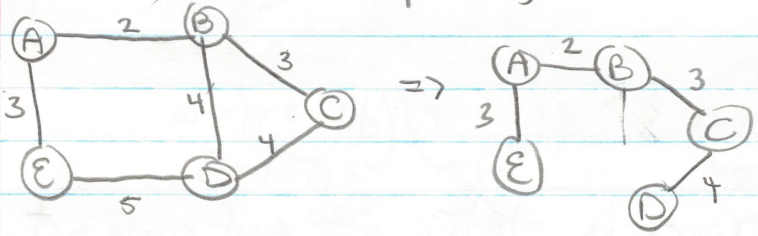
Hilroy

if min spanning tree is given with missing edge values



=> Must be smallest Value possible when comparing connecting edges

Ex. Find min cost spanning Tree

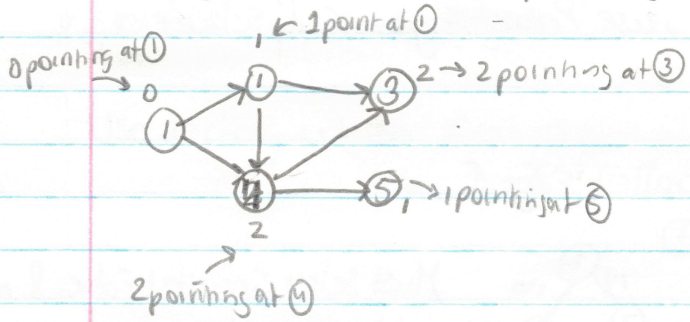


* Could have diff trees but will have same min cost *

Cost = 12

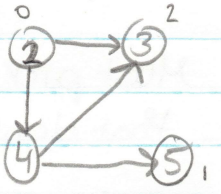
Topological Sort

Indegree: # of nodes pointing to it



Now Find indegree 0's: 1

- Now deleting that node & all its edges pointing out of it
- Put all new indegrees of all vertexes



- Now repeat this process: 1, 2, ...

```
BFS Algorithm (TreeNode root) {
  Queue<T> q = new LinkedList<>(); List = newLinkedList<>();
     q.enqueue(root);
  while (!q.isEmpty() ){
     TreeNode<E> current = q.dequeue()
       list.add(curren.item ) => Toreverse addtostart
     if (current.left ≠ null) q.enqueue(current.left)
     if(current.right ≠ null) q.enqueue(current.right)
while - 3
     for(E item: List)print(E)
```

DFS-Algorithm

Hilroy

Red-Black tree

1: A node is either red or black
2: The root and leaves (NIL) are black
3: If node is red, then its children are black
4: All paths from a node to its (NIL) descendants contain the same number of black nodes.

Extra Notes:
- Nodes require a storage bin to keep track of color
- Shortest path: all black nodes
- Longest path: alternating red and black

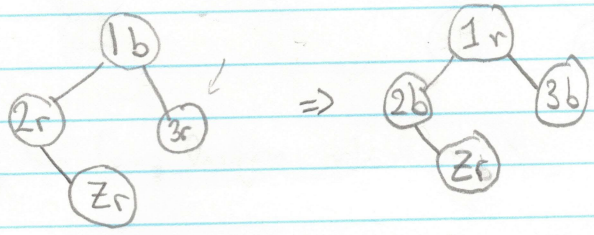
Time complexity:

Search	O(logn)
Insert	O(logn)
Remove	O(logn)

Strategy:
1. Insert z and color it red
2. Recolor and rotate nodes to fix violation

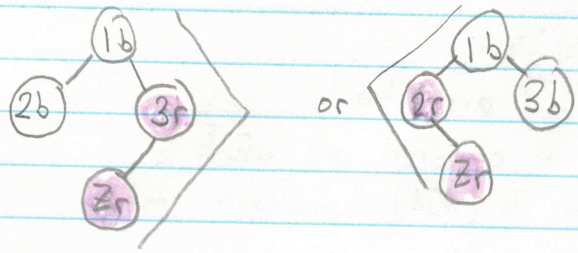
case 0: z = root
↳ color it black

case 1: z.uncle = red

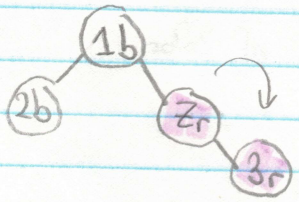


↳ Recolour z parent, grandparent, and uncle

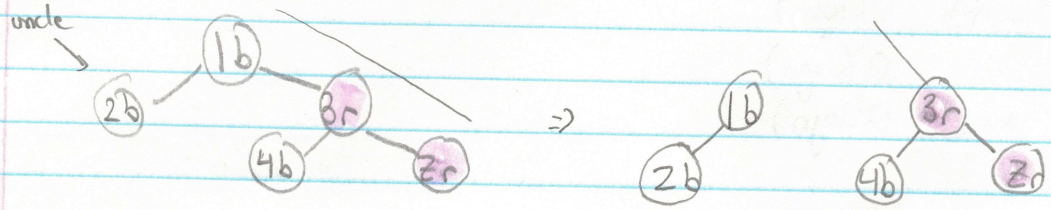
case 2 : z.uncle = black (triangle)



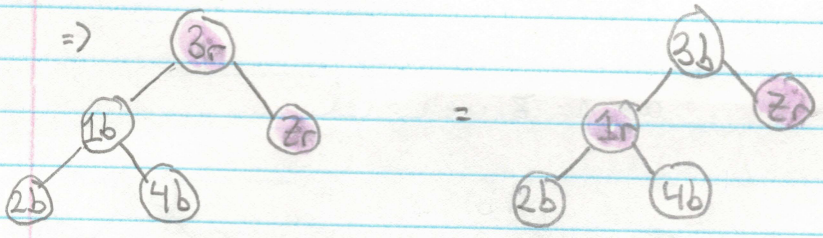
↳ Rotate z parent



case 3 : z.uncle = black (line)

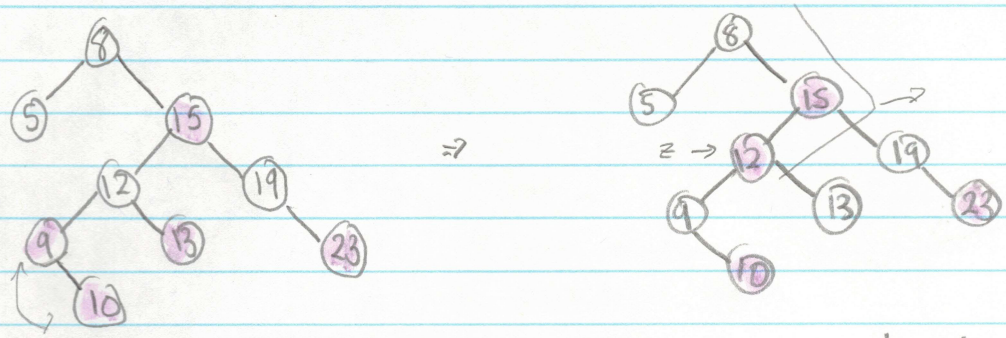


↳ Rotate z grandparent in opposite direction of z.



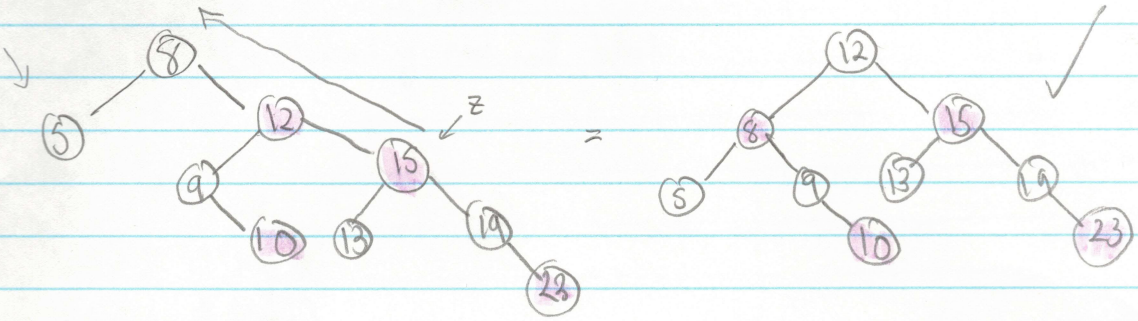
↳ then Recolor original parent and grandparent

Example:



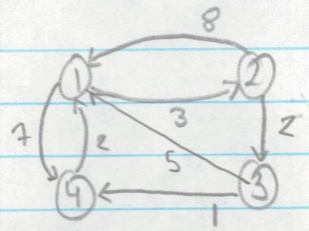
collision → case 1: z.uncle Red ⇒ Recolor parent, grandparent, and uncle

next collision case 2: z.uncle = black (triangle)
↳ Rotate in direction of arrow



case 4: z.uncle = black (line)
↳ Rotate opposite direction of z
↳ Recolor original parent + grandparent

Floyd Warshall Algorithm



$$A^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & \infty \\ 3 & 5 & \infty & 0 & 1 \\ 4 & 2 & \infty & \infty & 0 \end{array}$$

$$A^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & \infty & 0 \end{array}$$

$$A^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 5 & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & 7 & 0 \end{array}$$

$A^0[2,3]$ $\quad A^0[2,1] + A^0[1,3]$

$2 \quad < \quad 8 + \infty$

remains 2

$A^0[2,4]$ $\quad A^0[2,1] + A^0[1,4]$

$\infty \quad > \quad 8 + 7 = 15$

$A^1[1,3]$ $\quad A^1[1,2] + A^1[2,3]$

$\infty \quad > \quad 3 + 2$

$A^0[3,2]$ $\quad A^0[3,1] + A^0[1,2]$

$\infty \quad > \quad 5 + 3 = 8$

$A^1[1,4]$ $\quad A^1[1,2] + A^1[2,4]$

$7 \quad < \quad 3 + 15$

Formula $= A^k[i,j] = \min \{ A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j] \}$

The code:

```
for (k=1; k<=n; k++)
  {
    for (i=1; i<=n; i++)
      {
        for (j=1; j<=n; j++)
          A[i,j] = min (A[i,j], A[i,k] + A[k,j])
```

Fibonacci Heap

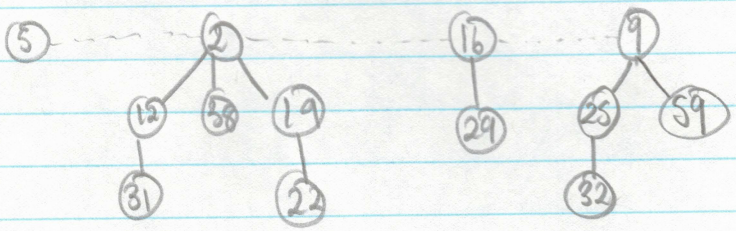
Data structure that implements a mergeable heap

1: make-heap : creates an empty heap
2: insert : inserts a node
3: minimum : returns the minimum node
4: extract-min : deletes the minimum and returns it
5: union : merges two heaps.
6: decrease_key : decreases the key for a node, with a value that is less than or equal to the current node
7: delete : deletes a node

time complexity:

operation #	Binary heap	Fibonacci heap	Linked List
make-heap	$O(1)$	$O(1)$	
insert	$O(\log n)$	$O(1)$	$O(1)$
minimum	$O(1)$	$O(1)$	$O(n)$
extract-min	$O(\log n)$	$O(\log n)$	$O(n)$
union	$O(n)$	$O(1)$	$O(1)$
decrease-key	$O(\log n)$	$O(1)$	
delete	$O(\log n)$	$O(\log n)$	

collection of min-heap ordered trees

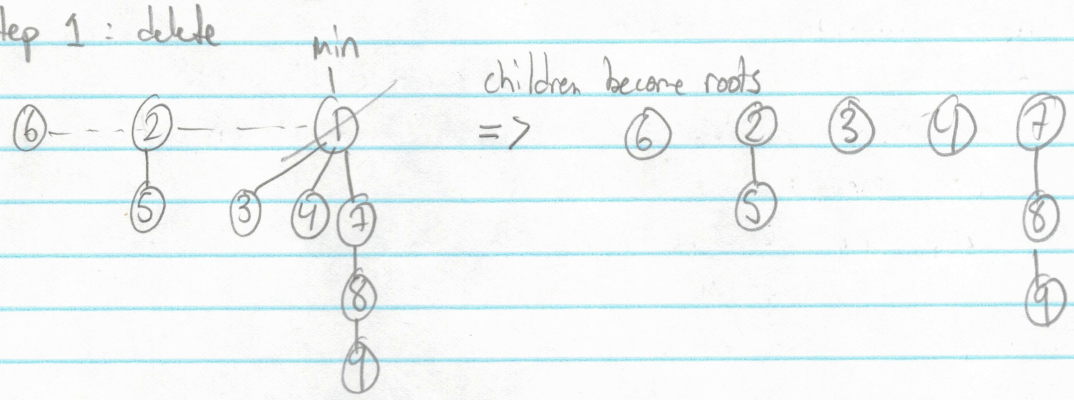


Hilroy

1- Find minimum operation: We have a pointer to it

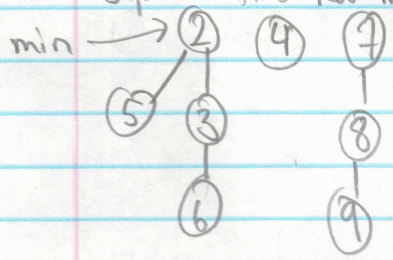
2- Delete Minimum:

Step 1: delete



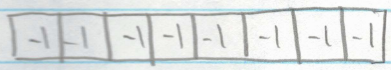
Step 2: Roots of same degree get stacked

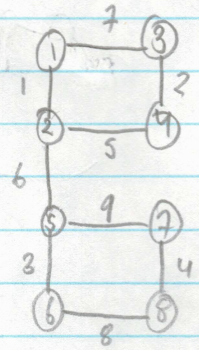
Step 3: find new min O(logn)



Disjoint Sets

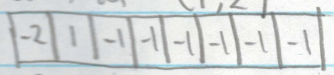
$U = \{1, 2, 3, 4, 5, 6, 7, 8\}$





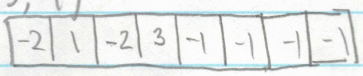
→ negative indicates it is a parent
→ value indicates number of nodes inside the graph

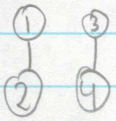
start at (1,2)

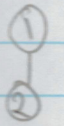


→ find 1 and 2 ⇒ both -1, different sets, we perform a union.
→ Positive value points to parent

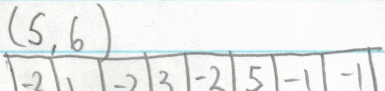
⇒ (3,4)

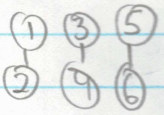




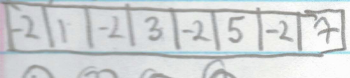


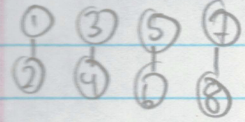
⇒ (5,6)



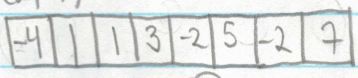


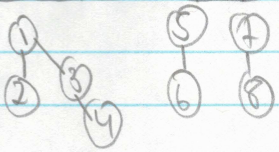
(7,8)



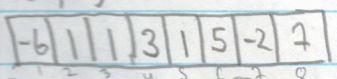


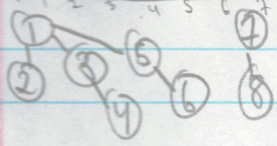
(2,4)





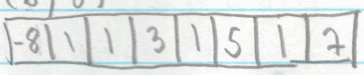
(2,5)

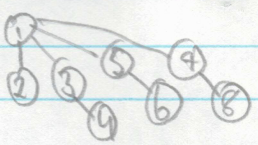




(1,3) → same parent → forms a cycle → ignored.

(6,8)

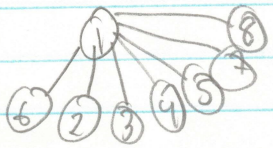




(5,7) → same parent → cycle

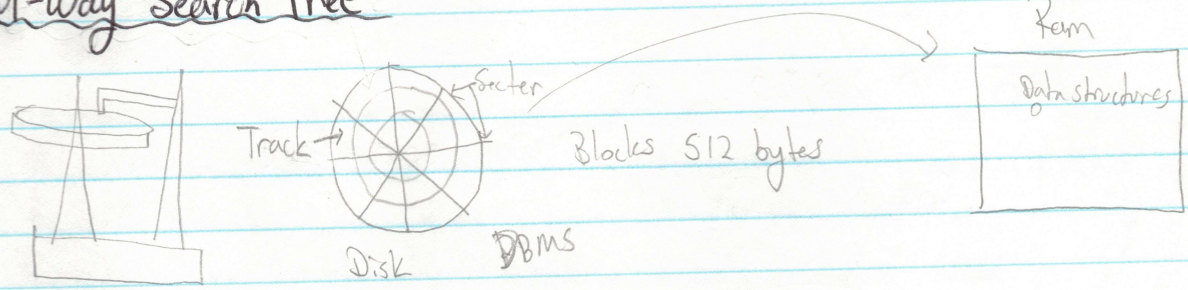
collapsing find:

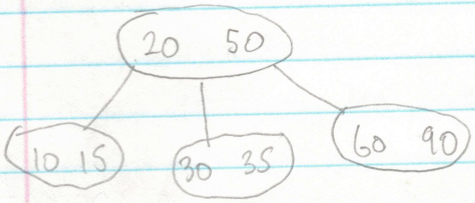
→ If we know that a node deeper down the tree belongs to a root, collapse it to the root:



→ this will make the search faster.

M-Way Search Tree

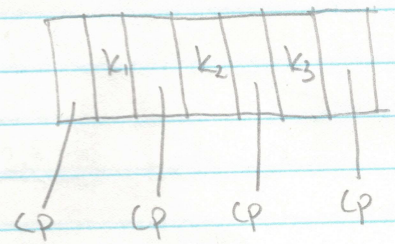




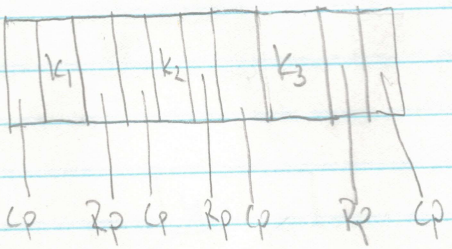
2 keys
3 children
] 3 way search tree

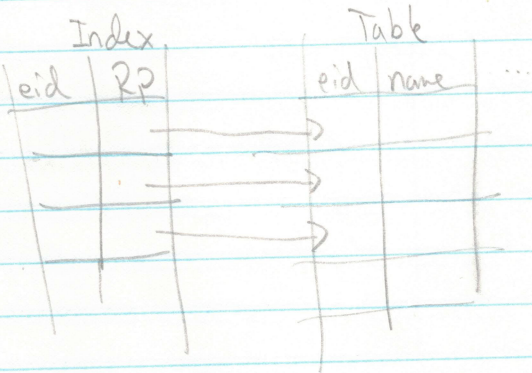
m → children
m-1 → keys

node for 4 way search tree:



we can use it to make indexs

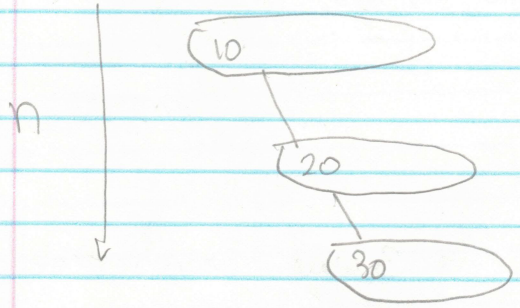




Insertion:

Keys : 10, 20, 30 .... n

10 way search tree



Problem: No control over insertion so we can have a height of n with n- keys, which is bad. Similar to linear search

This is why we use B-Trees

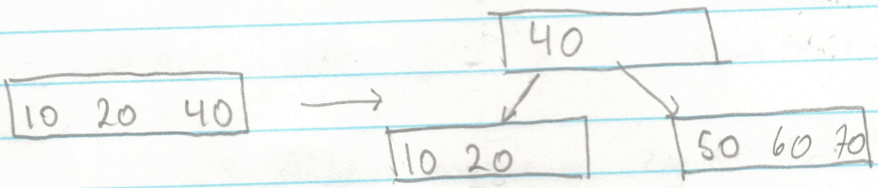
B-Trees

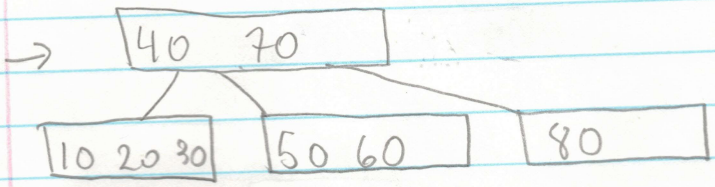
1. A node can have a maximum of m children (i.e. 3)
2. A node can contain a maximum of (m-1) keys (i.e. 2)
3. A node should have a minimum of $\lceil m/2 \rceil$ children (i.e. 2)
4. A node (except root) should contain a minimum of $\lceil m/2 \rceil - 1$ keys (i.e. 1)

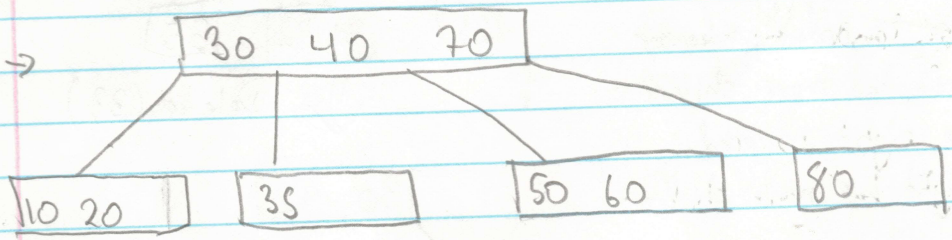
$m = 4$

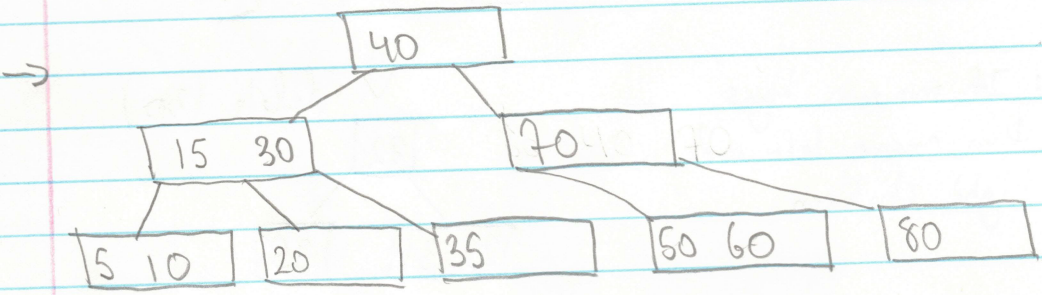
Keys: 10, 20, 40, 50, 60, 70, 80, 30, 35, 5, 15

Insertion:





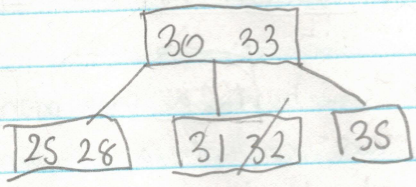


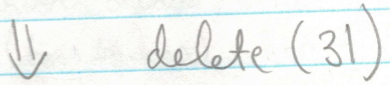


if $m=6 \Rightarrow t=3$ $\quad min = t-1 = 2$ $\quad max = 2t-1 = 5$

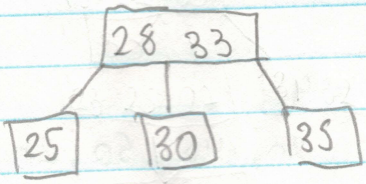
Deletions

case 1: Simple delete from leaf node
→ simply remove it

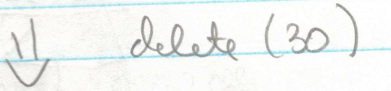




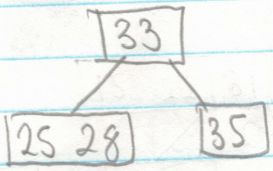
case 2: deletion violates minimum number of keys: We borrow key from neighbor.



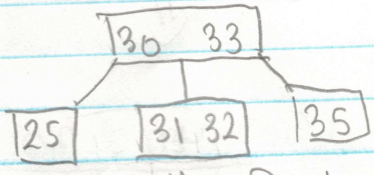
★ check left node first then check right node

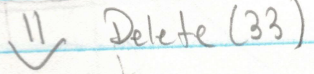


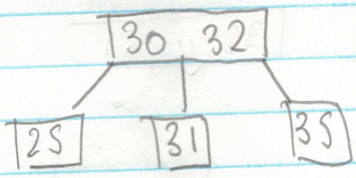
case 2b: if both siblings already minimum, merge with left or right sibling node



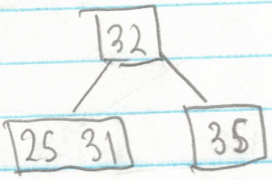
case 3: Internal node is deleted.
→ Choose in-order predecessor if left has more children
→ choose right IO pred. if right has more children



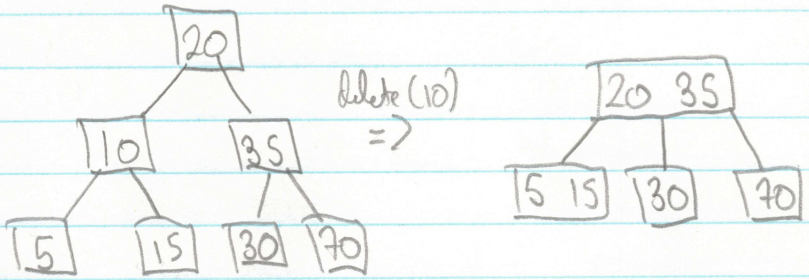




case 3b: If minimum keys then merge left and right children.



Case 4: Deletion internal node results in key less than minimum → height reduction



Time complexity:

Search = $O(\log n)$

Insert = $O(\log n)$

Delete = $O(\log n)$

B(2,5)

5 children maximum

2 children minimum per node.