Aug 27 Algorithms.

Bruce Maggs

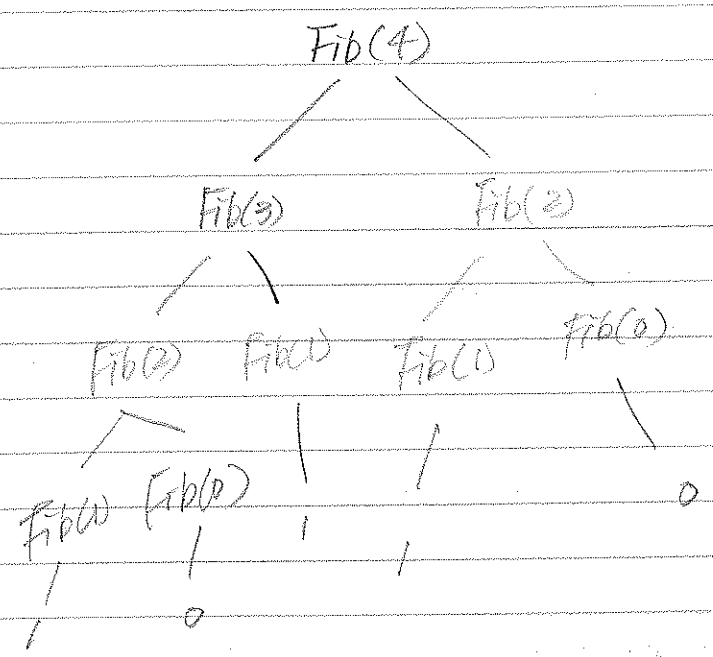
$$Fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ Fib(n-1)+Fib(n-2) & \text{if } n>1 \end{cases}$$

0 1 1 2 3 5 8 13 ... the ratio goes to $\phi = \frac{1+\sqrt{5}}{2}$
≈ 1.62

$$Fib(n) = \frac{\phi^n - \phi^{-n}}{\sqrt{5}} \approx \frac{\phi^n}{\sqrt{5}}$$

$$= O(\phi^n).$$

Use some algorithm to compute Fib(n): Recursive Algorithm



$$\text{Time} = T(n) = \begin{cases} T(n-1)+T(n-2)+O(1) \\ O(1) & \text{if } n<2 \end{cases}$$

leaves = # internal nodes + 1 $\Rightarrow$ # nodes $\geq Fib(n)$
$\leq 3Fib(n)-1$

$$T(n) = \Theta(Fib(n))$$
$$= \Theta(\phi^n)$$

We usually analyzise the size of the input when we analyzise time, but here we analyzise the magnitude.

size ⇒ # bits needed to represent input

Space =

$S(n) = O(n)$

Another:

```
fib_2 = 0
fib_1 = 0
for (i=2, i<n, i++) {
    fib_i = fib_1 + fib_2;
    fib_2 = fib_1;
    fib_1 = fib_i;
    return fib_i;
}
```

$T(n) = O(n)$

$S(n) = O(1)$

Problem: You keep doing the same thing over and over again

⇒ Memorizing ⇒ Save solutions to any subproblems that you solve

```
for(i = 0; i< MAX_N ; i++) {
    memo(i) = -1;
}
memo(0)=0;
memo(1)=1;

int fib (int n) {
    if (n<0) exit(0);
    if(memo(n) > -1){
        return memo(n)}
    else {
        memo(n) = fib(n-1)+ fib(n-2)
        return mem(n);
}
```

<MEMORIZATION>

$T(n) = O(n)$
$S(n) = O(n)$

— Dynamic Programming
- solve all subproblems
- use explicit schedule
- "Dynamic Optimality" = solution to any problem can be composed of solutions to subproblem

— Divide and Conquer
1. Divide the problem into smaller or easier subproblems
2. solve problems.
3. Use solutions to construct solution to original problem

DP=

```
for (i=0; i< MAX_N; i++) {
    Fib(i) = -1;
    Fib(0) = 0;
    Fib(1) = 1; }
for (i=2; i<n; i++) {
    Fib(i) = fib(i-2) + fib(i-1); }
```

$T(n) = O(n)$

$S(n) = O(n)$

Thursday Aug29th

Asymptotic Analysis:

Big O notation: upperbound
A function f(n) is O(g(n)) if g(n) multiplied by a constant factor is the upper bound of f(n).

T(n)=O(log(n)): binary search
T(n)=O(n^2): borderline

Space often matters more than time.
We want algorithms with linear time and space requirements.

Omega: lowerbound
theta: both upper and lower bound (tight bound)

Dynamic Programming:

1)Solve all subproblems (identify the subproblems)
2)Develop a schedule for solving the subproblems (usually the smallest and easiest first, usually bottom-up)

Knapsack:

MAX_WEIGHT--Capacity of the knapsack
n items 1…n
V[i] the value of item i
W[i] the weight of item i
Goal: maximize value without exceeding the MAX_WEIGHT (subject to the weight constraint)

S[i,j]=max value possible possibly using items 1,2,…i with total weight at most j. (We want to get S[n, MAX_WEIGHT])

```
for (k=0; k<=MAX_WEIGHT;k++){
    S[0,k]=0;
}

for (l=1;l<=n;l++){
    for(k=0;k<=MAX_WEIGHT;k++){
        if (k<W[l]){
            S[l,k]=S[l-1,k];
        }else{
            if (V[l]+S[l-1,k-W[l]]>S[l-1,k]){
                S[l,k]=V[l]+S[l-1,k-W[l]];
            }else{
                S[l,k]=S[l-1,k];
```

```
            }
        }
    }
}
```

To reconstruct the solution: see whether you use a previously constructed solution and mark.

Tuesday Sep 3rd CS330

Divide and Conquer

1. Divide the problem into simpler subproblems. (Small problems of the same form)
2. Solve the subproblems. (Usually recursively)
3. Combine the solutions to the subproblems to form the solution to the original problem.

Express Running Time using a Recurrence:

$$T(n) = T_{divide}(n) + T_{solve}(n) + T_{combine}(n)$$

$$\Downarrow$$

Often $a \cdot T\left(\frac{n}{b}\right)$

a: # of subproblems

b: size of subproblems

Mergesort:

Goal = Sort an array of items.

Input: | 3 8 4 1 | 2 5 7 8 |

| 1 3 4 8 | | 2 5 7 8 |

Merge = ↑→↑ ↑→↑

| 1 2 3 4 5 7 8 8 |

$T_{Merge}\left(\frac{n}{2}\right) = O(n)$

$S_{Merge}\left(\frac{n}{2}\right) = O(n)$

✡ Running Time of Merge Sort:

$T(1) = O(1)$

$T(n) = T_{divide} + T_{solve} + T_{combine}$

$= O(1) + 2T\left(\frac{n}{2}\right) + \underbrace{O(n)}_{\text{Merge}}$

Master Recurrence Theorem.

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ 2T(\frac{n}{2}) + O(n) \end{cases}$$

☆ Theorem: $T(n) = \begin{cases} aT(\frac{n}{b}) + O(n^c), & n>1 \\ O(1), & n=1 \end{cases}$

a, b, c are fixed constants. If $c > \log_b a$ then $T(n) = O(n^c)$

If $c < \log_b a$, then $T(n) = O(n^{\log_b a})$

If $n^c = \Theta(n^{\log_b a} \cdot \log^k n)$, $k \geq 0$, $k \in I$
then $T(n) = O(n^c \log^{k+1} n)$

In our case, $a=2$, $b=2$, $c=1$, $k=0$

$$n^c = n^1 = n^{\log_b a} \cdot \log^0 n = n$$
$$\Rightarrow T(n) = O(n \log n) \quad \text{(OK)}$$

Sorting requires $\Omega(n \log n)$ comparison $\Rightarrow$ Mergesort is optimal in terms of time, but requires linear extra space.

Brutal force:

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

Unroll $\Rightarrow$
$$= 2(2T(\tfrac{n}{4}) + \tfrac{n}{2}) + n$$
$$= 4T(\tfrac{n}{4}) + 2n$$
$$= 4(2T(\tfrac{n}{8}) + \tfrac{n}{4}) + 2n$$
$$= 8T(\tfrac{n}{8}) + 3n$$
$$\cdots$$
$$= n \cdot T(1) + (\log_2 n) \cdot n$$
$$= n(\log_2 n + 1)$$

$T(n)$ $\qquad n$

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

$T(\frac{n}{4})$ $\quad T(\frac{n}{4})$ $\quad T(\frac{n}{4})$ $\quad T(\frac{n}{4})$ $\qquad \frac{n}{4} \times 4$

$\vdots$

$T(2)$ $\qquad \frac{n}{2} \times 2$

(n at each level)

$T(1)$ $\quad T(1)$

# of leaves? $2^{\#\text{levels}} = 2^{\log_2 n} = n$

# of levels $= \log_2 n$

Time $\Rightarrow n \cdot \log_2 n$

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

$T(n)$ $\qquad n^2$

$T(\frac{n}{2})$ $\quad T(\frac{n}{2})$ $\qquad 2 \cdot (\frac{n}{2})^2 = \frac{n^2}{2}$

$T(\frac{n}{4})$ $\qquad 4 \cdot (\frac{n}{4})^2 = \frac{n^2}{4}$

$\vdots$

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

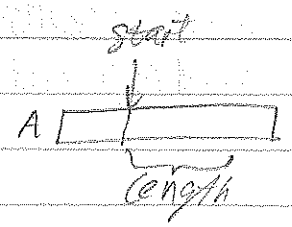
$n(T(1)) = n$

$T(1)$

Pseudo Code For Merge:

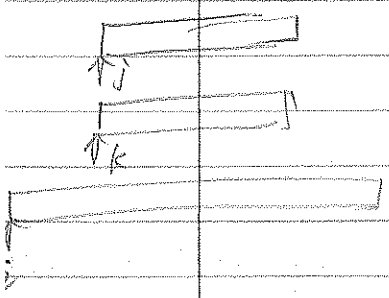
global array A ← to be sorted
global array B ← scratch space

```
void mergesort (int start; int len) {
    if (len == 1) return;
    int half = len/2
    int middle = start + half
    mergesort (start, half);
    mergesort (middle, len-half);
    merge (start, half, middle, len-half);
    return;
}
```




```
merge (start1, len1, start2, len2) {
    int j=0; int k=0;
    for (int i=0; i< len1+len2; i++) {
        if (j==len1) { B[i] = A[start2+k]; k++}
        else if (k==len2) { B[i] = A[start1+j]; j++};
        else if (A[start1+j] < A[start2+k]) {
                B[i] = A[start1+j]; j++}
        else {
                B[i] = A[start2+k]; k++}
    }

for (i=0; i< len1+len2; i++) { A[start1+i] = B[i]}
```



Sep 5th Thursday

Master Recurrence Theorem

$$T(n) = aT(\tfrac{n}{b}) + n^c \log^k n$$

Case 1. $c > \log_b a \Rightarrow T(n) = O(n^c)$

Case 2. $c < \log_b a \Rightarrow T(n) = O(n^{\log_b a})$

Case 3. $T(n) = aT(\tfrac{n}{b}) + n^c \log^k n$

$\Rightarrow \log_b a = c$, then $T(n) = O(n^c \log^{k+1} n)$

Radix Sort

- MSD = Most Significant Digit first (Divid and Conquer
- LSD = Least Significant Digit first

Goal = Sort items with binary representations.

e.g.: 4-bit integers

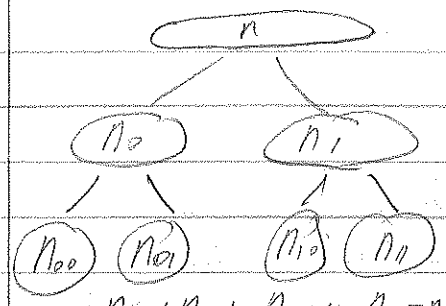
Int =					
2 = 0010	0010	0010	0001	000
7 = 0111	0111	0001	0000	000
1 = 0001	⇒ 0001	⇒ 0000	⇒ 0010	⇒ 001
13 = 1101	0000	0111	0111	011
0 = 0000	1101	1001	1001	100
9 = 1001	1001	1101	1101	110

Divide and Conquer =

1. divide = sort according to first digit
2. Conquer = recursively sort
   (1) Keys that start with 0
   (2) Keys that start with 1
3. Combine : Nothing to do

- count #0 vs #1
- copy A into B with keys starting with 0 ahead keys starting with 1

Recursion Tree:



$n_{00}+n_{01}+n_{10}+n_{11}=n$

$-O(n)$

$n_0+n_1=n \quad -O(n_0)+O(n_1)=O(n)$

$-O(n)$

levels = # digits per key.

$\Rightarrow T_{total} = O(n \cdot k)$

digits

(array)

```
void msd-radix (int *A, int n, int k, int pos) {
        if (n<=1) {return}; // Base case
        int zeros = 0;
        for (int i=0; i<n; i++) {
2 ←             zeros = zeros + (1-((A[i] >>(k-1)-pos) & 1));}
        int first_zero=0;
        int first_one= zeros;
        int *B = malloc (n * sizeof (int));

        for (int i=0; i<n; i++) {
            if ((A[i] >>(k-1)-pos) & 1 == 0) {
                B[first_zero] = A[i];
                first_zero ++;
            }else {
                B[first_one] = A[i];
                first_one ++;

        for (int i=0; i<n; i++) {
            A[i]=B[i];
        }
```

```
msd_radix(A*, zeros, k, pos+1);
msd_radix(A*+zeros; n-zeros; k, pos+1);
```
// recursive steps.

LSD Radix Sort:

Sorted according to the two LSDs

0010	0010	0000	0000	0000
0111	0000	0001	0001	0001
0001	0111	1101	1001	0010
1101	0001	1001	0010	0111
0000	1101	0010	1101	1001
1001	1001	0111	0111	1101

⇓ ⇓

sort according to the LSD first

sort according to 2nd LSD must be stable

— Stable sorting algorithm: preserves order of keys with same value.

$T(n) = O(k \cdot n)$

```
void lsd-radix (int *A, int n, int k) {
    for (int pos=k; pos>=0; pos--) {
        for (int i=0; i<0; i++) {
            zeros = zeros + (1-(A[i] >> (k-1)-pos & 1));}
        int first_zero=0; int first_one= zeros;
        int *B = malloc (n * sizeof(int));
        for (int i=0; i<n; i++) {
            if( (A[i] >> (k-1)-pos) & 1)==0 ) {
                B [first_zero] = A[i];
                first_zero++;
            }else{
                B[first_one] = A[i];
                first_one ++;
```

Sep 10th Tuesday.

1. Quicksort
2. Quickselect
3. Linear-time deterministic selection.

— All three are divide-and-conquer algorithms

Quicksort

1. Pick a "partition" element
   (e.g. a random element, first element, median)

2. Separate the elements <= partition from those > partition

3. Sort the two groups separately.

- In memory / in place.
- Not stable

next: # of elements <= pivot so far

0	1	2	3	4	5	6	7	8	
3	2	8	4	8	6	5	6	2	A

pos = 6

3	2	8	4	8	6	2	6	5

Step 2: return # of elements <= A[pos]

```
int partition (int *A, int n, int pos) {
    if (n=0) { return -1; }
    if (n=1) { return 1 }
    pivot = A[pos]; next = 0;
    swap (A, pos, n-1);
    for (int i=0; i<n; i++) {
        if (A[i] <= pivot) {
            swap (A, i, next);
            next++; }
    return next}
```

```
swap (A, i, j) {
    int temp = A[i]
    A[i] = A[j];
    A[j] = temp; }
```

```
quicksort (A, n){
    If (n<2) {return;}
    int pos = rand (0, n-1);
    int middle = partition [A, n, pos];
    quicksort (A, middle-1);
    quicksort (A+middle, n-middle);
}
```

- If everything works well:

$$T(1) = O(1)$$

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

- Worst case:

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

- Bad idea to pick 1st element as pivot.

Selection = Given a list of n elements, identify k^{th} largest.

— Observation = Easy to reduce to sorting.
Idea = sort A, pick A[k].

— Key Observation = If we can determine the order of any one element, we can eliminate some other candidates for k^{th} largest.

— Assume without loss of generality that all elements are distinct.

Quickselection =

```
int QuickSelect (int *A, int n, int k) {
        if (n=0) {return -1;};
        int pos = rand(0, n-1);
        int middle = partition [A, n, pos];
        if (middle-1 = k) {return A[middle-1];}
        if (middle-1 < k) {return QuickSelect (A+middle, n-middle,
                                                k-middle);}
        if (middle-1 > k) {return QuickSelect (A, k-middle, k
        }
```

— If all goes well =

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

— Worst Case =

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

Deterministic Linear-Time Selection

1. Break n elements into group of 5, and sort each group = $O(n)$ time.

2. Find recursively the median of $\frac{n}{5}$ medians.

3. Determine the order of median of medians:

$$\frac{n}{4} < \text{order} < \frac{3n}{4}$$

$O(n) \Rightarrow$ compare with every element.

4. Recursively discard at least $\frac{n}{4}$ elements.

Recurrence: $T(1) = O(1)$

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{3}{4}n\right) + O(n)$$

$$= O(n)$$

Sep 12 Thursday

Searching

1. Binary Search
   - tail recursion.

2. Binary Search Trees

3. Balanced Search Trees
   - Top-down 2-3-4 Trees

Binary Search

```
int search (int *A, int l, int r, int key) {
    if (l>r) {return -1};
    int mid = (l+r)/2;          /* C runs down */
    if (A[mid] == key) {return mid};
    if (A[mid] < key) {return search(A, mid+1, r, key)};
    else {return search(A, l, mid-1, key)}
}
```

Bruce Maggs

Grand Father (LAW) (NO LAW/CS or DINNER)

(LAW) Father Mother

(CS) Brother (LAW)

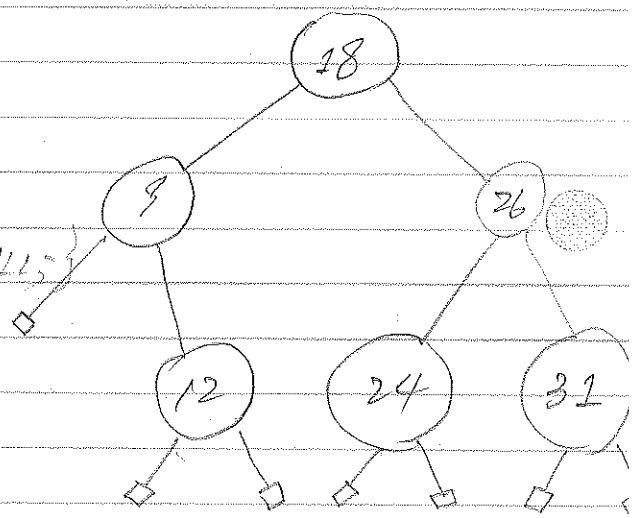
- Tail recursion: after recursive call, just return result without using any local variables.

Iteratively:

```
int search (A, l, r, key) {
    if (l>r) {return -1;};
    while (l<=r) {
        mid = (l+r)/2;
        if (A[mid] == key) {return mid;}
        if (A[mid] < key) {l = mid+1;}
        else r = mid-1;
    }
    return -1;
}
```

Problem = Hard to Insert / Delete / Replace

Binary Search Trees




```
node* Search (node *root, int key) {
    if (root == NULL) {return NULL;}
    if (root->key == key) {return root;}
    if (root->key < key) {
        root = root->right;
    } else {
        root = root->left;
    }
    return search (root, key);
}
```

- Each node has 2 children including dummy children.
- All keys in left subtree are < key at root; all keys in right subtree are > key at root. True for every sub tree.

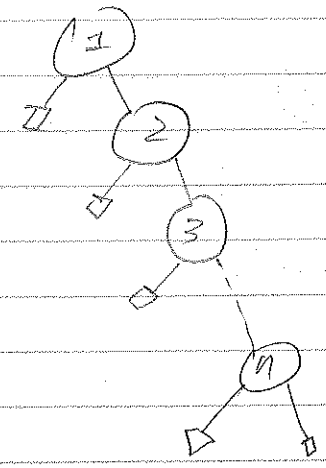
```
typedef
struct nodes {
    node* left;
    node* right;
    int key; } node;
```

```
node* insert (node* root, int key) {
    if (root->key > key) {
        if (root->left == NULL) {
            root->left = malloc (sizeof(node));
            root = root->left;
            root->key = key;
            root->left = NULL;
            root->right = NULL;
            return root;
        } else {return insert (root->left, key);}
    } RIGHT;
}
```

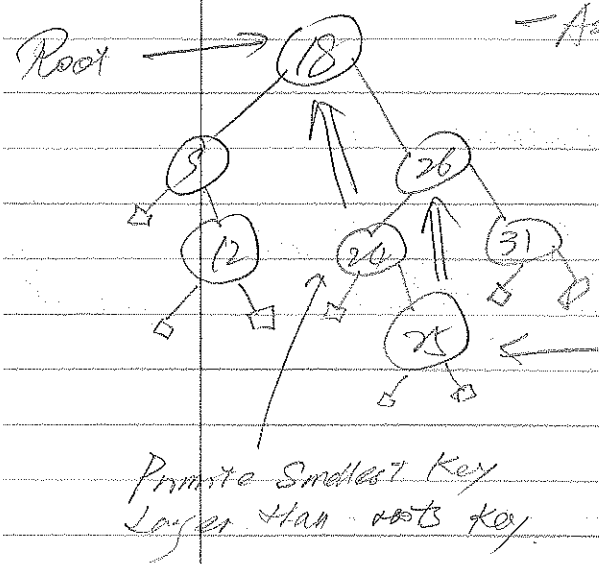
insert 1, 2, 3, 4, ...

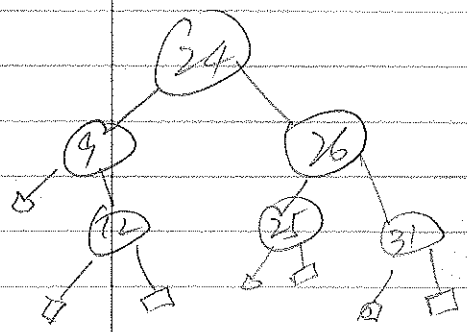
Depth of tree = n-1

$\boxed{\text{Not Balanced}}$



Deletion





- Assume we always delete root.
- Promote node with smallest key in right subtree
- That node has no left child.
- New left child is left child of old root
- New right child is right child of old root
- New left child of new root's old parent is new root's old right child.

Tuesday Sep 17

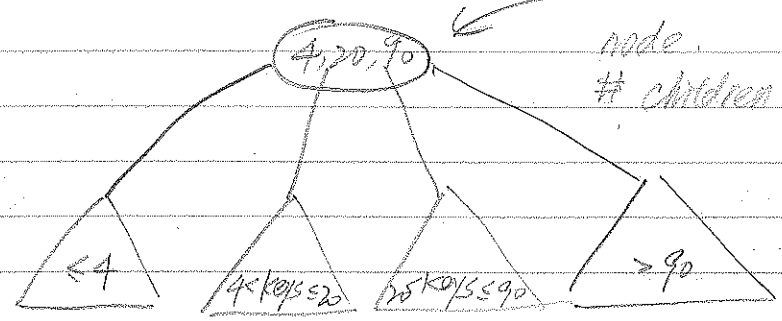
Balanced Search Trees

- top down 2-3-4 trees
  - worst case $O(\log(n))$ time per search, insert, delete operations.

- Splay Trees
  - Binary search trees
  - $O(\log(n))$ amortized time per search, insert, delete
    n operations take $O(n \log(n))$ time in worst case

Top-Down 2-3-4 Trees

Store 1, 2, or 3 keys in each node.
children = # keys + 1



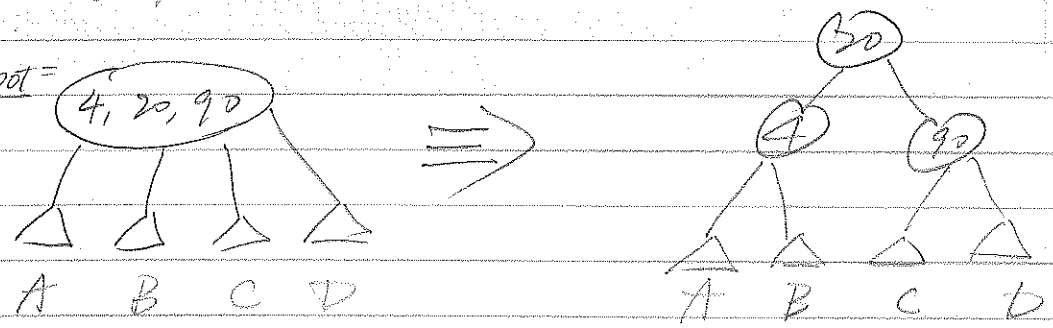
- All leaves are at same distance from root.
  If there're n keys, how deep can such a tree be?
  Worst case = all nodes have degree 2, depth = $O(\log(n))$

⇒ Search takes $O(\log n)$ time

Insert =

1. Search top-down for where new key should be added to tree
2. Break any nodes with 4 children encountered along path

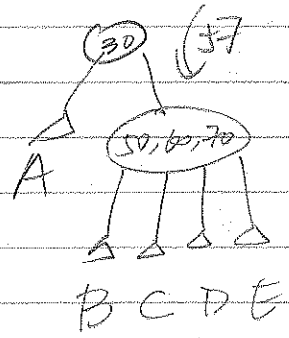
- At Root =



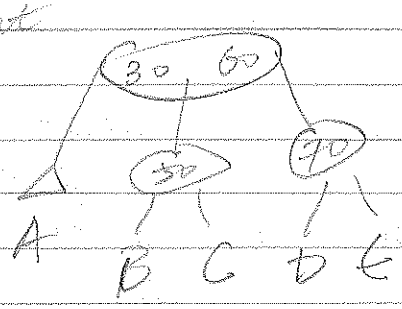
→ Distance to all leaves increased by 1

- 4-children node not at root = Two cases

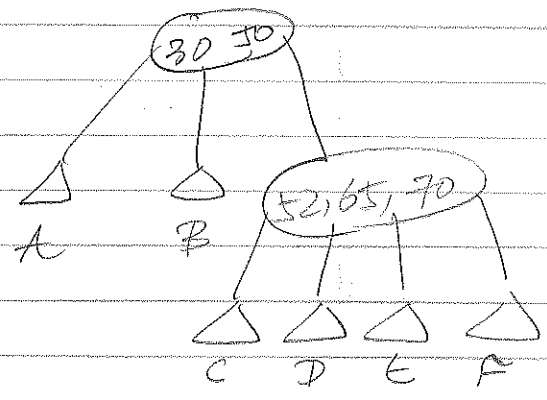
— parent has 2 children

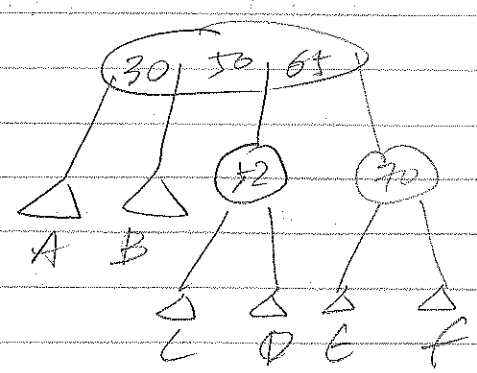


promote middle key to parent



— parent has 3 children



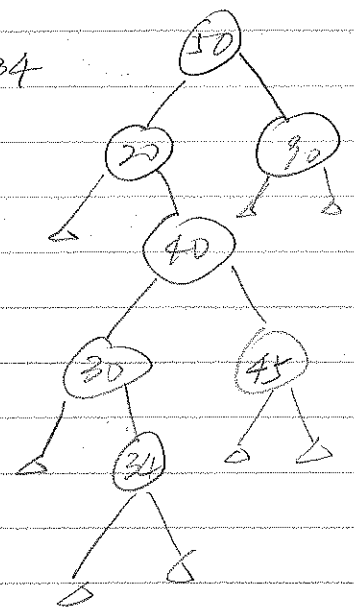


Splay Trees

- Binary search trees
- Every insert, delete, search, starts with a Splay operation

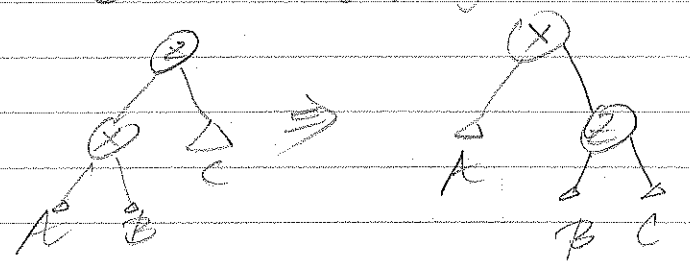
Splay (T, x)
- Move x to root if x is in the tree
- Move predecessor of x to root if x is not present
- First look for x (or its predecessor) using standard binary search tree search.

① X=34

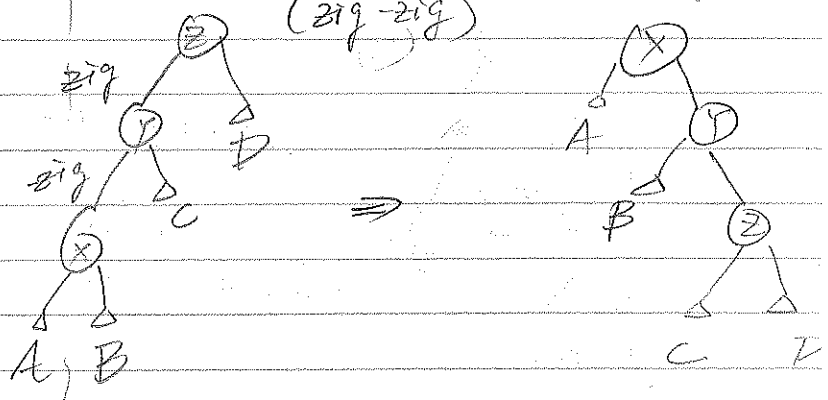


Use rotations to bring x to root (bottom-up)

① x is a child of root



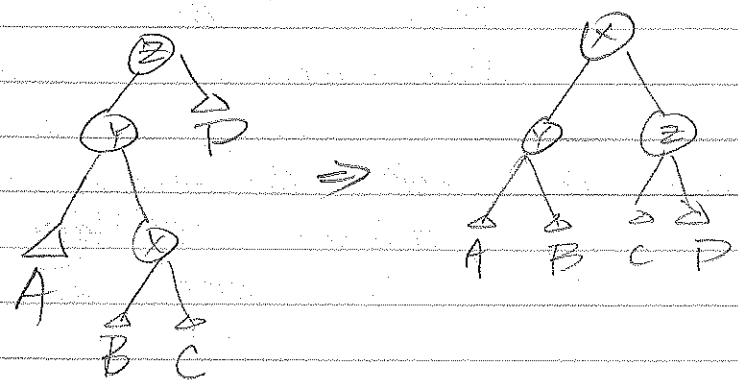
② x is the left child of the left child (zig-zig)



T = O(1)

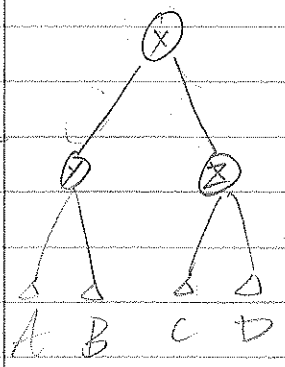
For each rotation

③ zig-zag

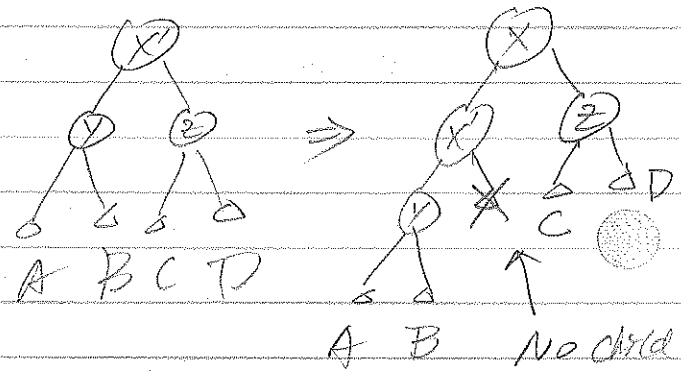


After Splay, to delete X =

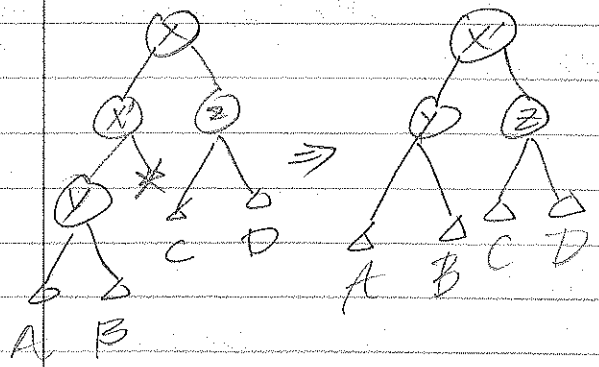
X has no child



After Splay, to insert X =



1. Bring X to root
2. Bring X' to root of left subtree



To prove n operations take $O(n \log n)$ time we use a potential function argument.

$\Phi(t)$ = potential of tree $T \geq 0$, $\Phi(\text{empty tree}) = 0$

Accounting Scheme

For each insert, delete, search =

1) count real work performed

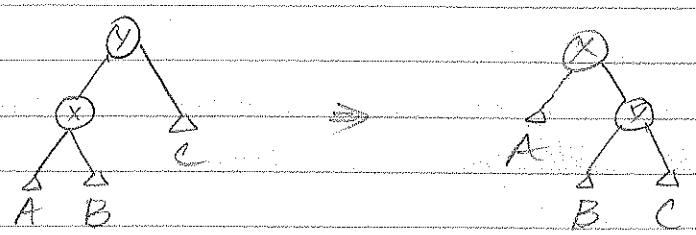
2) count change in tree potential Φ.

Why?

— It's easier to count both together than just real work.

Sep 19th Thursday

1. Amortized Analysis of Splay Trees
2. Heaps.



Before search, insert, or delete, Splay a key x to the root.

$\Phi(T)$ potential of tree T.
Amortized time t for one operation =
actual cost of the operation t
+
change in $\Phi(T)$.

Amortized cost of sequence of n operations

$$\underbrace{t_1 + t_2 + t_3 + \cdots + t_n}_{\text{Cost of performed rotations}} + \underbrace{(\Phi(T_1) - \Phi(T_0)) + (\Phi(T_2) - \Phi(T_1)) \cdots + (\Phi(T_n) - \Phi(T_{n-1}))}_{\Phi(T_n) - \Phi(T_0)}$$

T_i = tree after i operations
t_i = actual cost of ith operation

could be positive or negative.

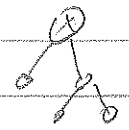
Actual cost Σt_i = total amortized cost + $\Phi(T_0) - \Phi(T_n)$

↑ Difficult to measure

↑ Easier

$O(n \log n)$ $O(n \log n)$

size(x) = # descendants of x, including x
(size of x's sub tree)

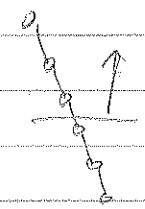


size(x) = 5

rank(x) = $\log_2$ size(x)

$$\Phi(T) = \sum_{x \in T} \text{rank}(x)$$

High $\Phi(T)$:



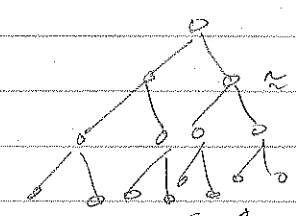
$\frac{n}{2}$ nodes above

size $\geq \frac{n}{2}$

rank $\geq \log_2 n - 1$

$\Phi(T) \geq \Omega(n \log n)$

Low $\Phi(T)$ =



log(n)

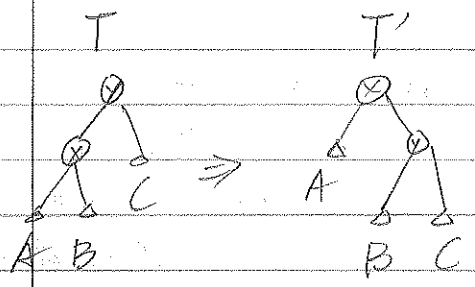
$\approx \log(n) - 1$

$\approx \log(n) - 2$

s = 1 rank = 0

$\Phi(T) \approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \cdots$

$\approx O(n)$



rank = r(x) (rotation)

Amortized time for rotation = 1 + $\Phi(T') - \Phi(T)$

actual cost

≤ 0

$= 1 + (r'(x) - r(x)) + (r'(y) - r(y))$

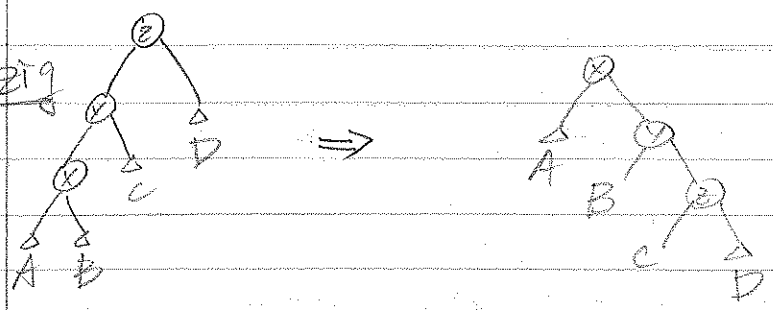
$\leq 1 + r'(x) - r(x)$

$\leq 1 + 3(r'(x) - r(x))$

Balanced?

MODIFIED NEXT PAGE

Zig-Zig



$\leq -r(x)$

$$ATR = 2 + \Phi(T') - \Phi(T)$$
$$= 2 + (r'(x) - r(x)) + (r'(y) - r(y)) + (r'(z) - r(z))$$
$$= 2 + r'(x) - 2r(x) + r'(z) + (2r'(x) - r(x) - r'(z))$$
$$\leq 3(r'(x) - r(x))$$

≥ 2

Show: $2r'(x) - r(x) - r'(z) \leq 2$ $\quad \geq 2$

$$= \log \frac{(size'(x))^2}{size(x) \cdot size'(z)} \geq \log 4 = 2$$

$size'(x) \geq s(x) + s'(z)$

Assume $size'(x) = s(x) + s'(z)$

$s(x) = s'(z) = \frac{1}{2} s'(x)$

Amortized Cost $= 3(r'(x) - r(x)) + 3(r''(x) - r'(x)) \cdots$
$+ 3(r_{root}(x) - r_{root-1}(x)) + 1$

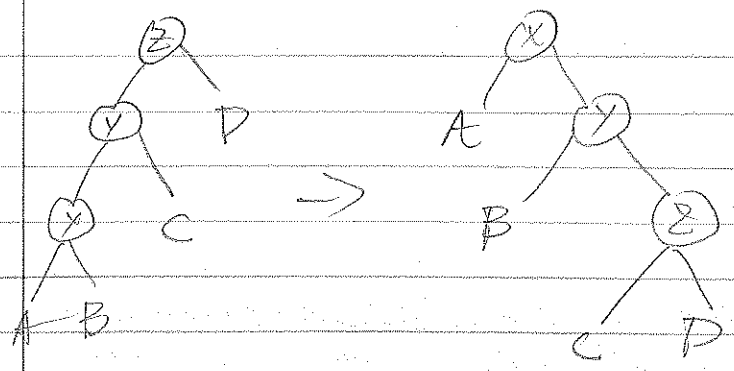
$= 3(r_{root}(x) - r_{old}(x)) + 1$

$\leq 3\log n + 1$

Total Amortized Cost $= 3n\log n + n$

Real cost could not be $\underbrace{n\log n}_{Max(\Delta\Phi)}$ off from amortized cost

MODIFIED NOTE:



$r(x) = \text{rank}(x) = \log_2 \text{size}(x)$

$s(x) = \text{size}(x)$.

$r'(x)$ = rank of x after rotation

Amortized Time For Zig-Zig

Actual cost

$= 2 + (r'(x) - r(x)) + (r'(y) - r(y)) + (r'(z) - r(z))$

$= 2 - r(x) + r'(y) - r(y) + r'(z)$ since $r(z) = r'(x)$

$\leq 2 - r(x) + r'(x) - r(x) + r'(z)$ since $r'(x) \geq r(x)$ and $r(x) \leq r(y)$

$= 2 + r'(x) - 2r(x) + r'(z)$

Now show:

$2 \leq 2r'(x) - r(x) - r'(z)$

so that amortized time

$\leq 2r'(x) - r(x) - \underline{r'(z)} + r'(x) - 2r(x) \underline{+ r'(z)}$

$= 3(r'(x) - r(x))$

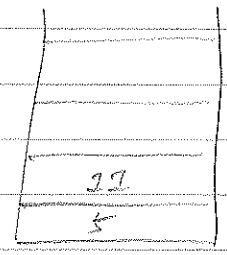
Tuesday | Sep 24th

Stack.

Operations:

- Push — ?
- Pop — O(1)
- Top — O(1)

Represent using an array:



size = 4
fixed sized array

— If the array is full:
Allocate new memory for an array twice as large, and then copy the old contents to the new array

— No worries about wasted space.

```
typedef struct {
        int size;
        int top;
        int *contents} stack;

void push (stack S, int new){
    if (top == size - 1) {
        int * new_contents = malloc (2 * size * sizeof(int));
        size = size * 2;
        for (int i=0, i< top, i++) {
            new_contents[i] = stack.content[i]; }
        free (stack. contents);
        stack. contents = new_contents; }
    ← top++;
    ← Stack. contends [top] = new;
```

$\Phi = 2 * top - size$

amortized operation cost for push =

(1) Stack not full =

— Actual cost = 1

— Change of potential = 2.

$1 + 2 = 3$

(2) Stack is full = $\underbrace{\overset{push}{1} + \overset{copy}{size}}_{\text{Actual Cost}} + \underbrace{2 - size}_{\Delta\Phi} = 3$

$$\sum_{i=1}^{n} T_{amortized}(O_i) = \sum_{i=1}^{n} \left(T_{actual} + \Delta\Phi\right)$$

$$= \sum_{i=1}^{n} T_{actual} + \Phi_{final} - \Phi_{initial}$$

$$\Rightarrow \sum_{i=1}^{n} T_{actual} = \sum_{i=1}^{n} \underbrace{T_{amortized}}_{3} + \underbrace{\underset{\overset{\shortparallel}{0}}{\Phi_{initial}} - \Phi_{final}}_{\leq 0}$$

$\Phi_{final} \downarrow$ $2 top - size \geq 0$ because $2 top \geq size$

$$\Rightarrow \sum_{i=1}^{n} T_{actual} \leq 3 \cdot n$$

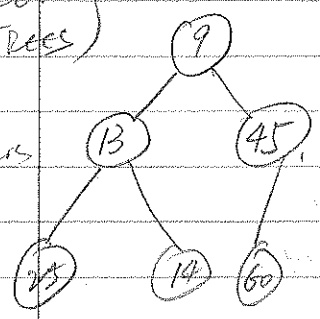
Heap = Full binary tree.

- Every level is full except possibly the lowest level.
- Lowest level is full from left to right.

* — Key root of any subtree ≤ all keys in subtree

(NOT ORDERED SEARCH TREES)

Do not store actual pointers



Operations = Insert, Remove-min, delete, change value, etc } $O(\log n)$ worst case, n is number of key

Can be represented using an Array =

a	9	13	45	25	14	60							
	0	1	2	3	4	5	6						

Rule = the left child of i is at $2 \cdot i + 1$

the right child is at $2i + 2$

the parent is at $\lfloor \frac{i-1}{2} \rfloor$.

Remove-Min =

(1) Record value of key at root

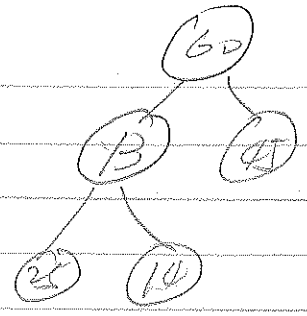
(2) Replace key at root with rightmost leaf key

(3) Remove rightmost leaf.

(4) Fix heap property =

If a key is greater than 1 or both children, swap with the smaller child

Fix Heap =




```
k=0,
v=a[k]                    k has a left child
while (2*k+1<n){
    j=2*k+1;              has a right child
                          right child smaller
    if (j+1<n && a[j+1]<a[j]) j++;

    if (v<=a[j]) break;
    else{a[k]=a[j]; k=j}
}
a[k]=v;
```