Course 1 Algorithm ToolBox

Week 2

Fibonacci Number

Fibonacci Number has rapid Growth Lemma:

$F_n \geq 2^{\frac{n}{2}} for\ n \geq 6$ Proof By induction

Base case n = 6, 7 (by direct computation) Inductive step

$$F_n = F_{n-1} + F_{n-2} \geq 2^{\frac{n-1}{2}} + 2^{\frac{n-2}{2}} \geq 2 * 2^{\frac{n-2}{2}} \geq 2^{\frac{n}{2}}$$

Close form of Fibonanci Number $F_n = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - \left(\frac{1-\sqrt{5}}{2}\right)^n)$

Idea 1: Recussion F(n) = F(n-1) + F(n-2)

Idea 2: for loop:

FibList(n):

Create An array F[0,…,n]

F[0] = 0; F[1] = 1

For i from 2 to n :

F[i] = F[i-1] + F[i-2]

Return F[n]

Greatest Common Divisor : 找最大公约数

Idea 1 : 从 1 for loop 直到找到

Idea 2: Implement the Euclidean Algorithm

Lemma : Let a′ be the remainder when a is divided by b then gcd (a,b) = gcd (a′,b)= gcd (a, b')

Proof: $a = a' + bq$ 如果 a , b 整除 d, 则a′ *除以b* 的余数也为 0

EuclidGCD(a,b):

If b = 0: Return a;

$a' = the\ remainder\ when\ a\ is\ divided\ by\ b$

Return EuclidGCD(b, a')

Takes about log (a,b) steps

BIG O Notation

Asymptotic runtimes: runtime scale with input size

$$\log n < \sqrt{n} < nlogn < n^2 < 2^n$$

Advantage of Big O Notation: 1. Clarify growth rate, 2. Clean up notation

Disadvantage of Big O Notation: 1. Using Big-O loses important information about constant multiples. 2. Big-O is only asymptotic.

如果 factor 100 是 bigger deal, $100n^2, n^2$

Big O Notation: f(n) = O(g(n)) (f is Big – O of g) or f<= g if there exist constants N and c so that for all $n \geq N, f(n) \leq c * g(n)$ (f grows slower than g)

Omega Notation: f(n) = Ω(g(n)) or f >= g if there exist constants N and c so that for all $n \geq N, f(n) \geq c * g(n)$ (f grows no slower that g)

small O Notation: For functions f,g : N-> R^+ we say that: f(n) = o(g(n)) or f< g if f(n) / g(n) -> 0 as n->∞ (f grows slower than g)

Theta Notation: f(n) = Θ(g(n)) or f ≈ g if f = O(g) and f = Ω(n) (f grows at the rate rate as σ)

O 是 upper bound, o 是 strictly upper bound， Ω 是 lower bound， Θ是 same rate

Logrithm 公式：

$$log_a(n^k) = k \log_a n$$

$$\log_a(nm) = \log_a n + \log_a m$$

$$n^{\log_a b} = b^{\log_a b}$$

$$\log_a n * \log_b a = \log_b n$$

Week 3 Greedy Algorithm

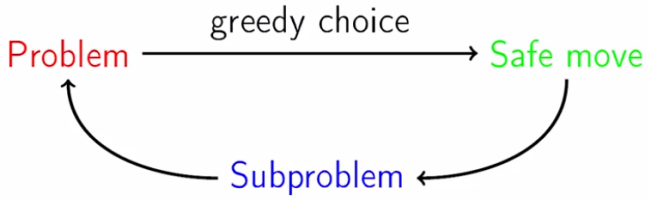
Greedy Algorithms

1. Reduction to subproblem
   - Make a first move
   - Then solve a problem of the kind
   - Smaller: e,g fewer digits, fewer fuel station problem

Safe Move : A move is called safe if there is an optimal solution consistent with the first move 第一个 move 是 safe 的，是跟 optimal 相关的，走了这步，就会走向 optimal

Not all first moves are safe

Often greedy moves are not safe.



保证第一个 move 是 safe move

- Make a greedy choice
- Prove that it is a safe move
- Reduce to a subproblem
- Solve the subproblem

Grouping Children

Problem: Many children came to a celebration. Organize them into the minimum possible number of groups such that the age of any two children in the same group differ by at most one year. Aslo want to minimize the groups

Naïve implementation : 试所有可能性的 group partitation 然后找到最小 group 的

MinGroups(C)

```
m ← len(C)
for each partition into groups
C = G1 ∪ G2 ∪ ··· ∪ Gk:
  good ← true
  for i from 1 to k:
    if max(Gi) − min(Gi) > 1:
      good ← false
  if good:
    m ← min(m, k)
return m
```

Lemma:

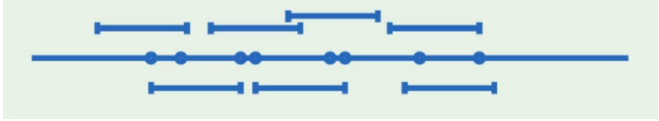
The number of operations in minGroup (C) is at least 2^n where n is the number of children in C (2^n *种分类*)

Proof

- Consider just partitions in two groups
- $C = G_1 \cup G_2$
- For each $G_1 \subset C$, $G_2 = C \setminus G_1$
- Size of C is n
- Each item can be included or excluded from G_1
- There are 2^n different G_1

Efficient Implementation： 把儿童的岁数换成点，把问题换成 the minimum number of segments of unit length needed to cover all the points

Example



Safe Move: cover the leftmost point with a unit segment with left and in this point

PointsCoverSorted($x_1, \ldots, x_n$)

```
R ← {}, i ← 1
while i ≤ n:
  [ℓ, r] ← [xi, xi + 1]
  R ← R ⋃ {[ℓ, r]}
  i ← i + 1
  while i ≤ n and xi ≤ r:
    i ← i + 1
return R
```

Efficient Algorithm with sored point, sorting O(nlogn)

running time O(n)

Fractional Knapsack

Problem : Input: weights w1, … Wn and values of items v1, …. Vn; capacity W Output: the maximum total value of fractions of items that fit into a bag of capacity W

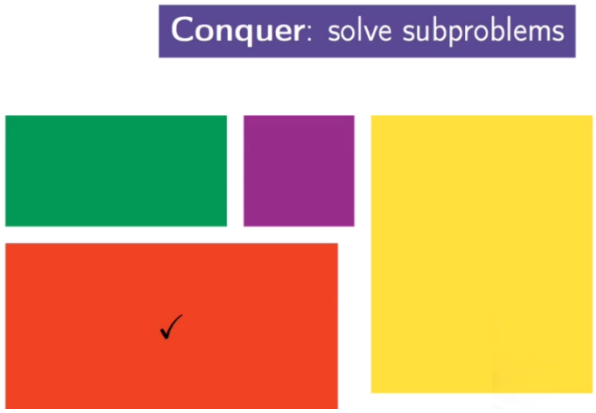
Efficient algoritm: 找每个 unit 哪个最值钱

Summary of Greedy Algorithm

1. Safe move
2. Prove safety
3. Solve subproblem
4. Estimate running time

Week 4 Divide and Conquer

Break down problem into a set of non-overlapping subproblems of the same original type, 必须 break down problem into the same type as 原来的， then solve problem independently (小的问题不能 overlapping， 也不能不一样的 type)



既然每个 subproblem 都是一样的，可以用 recursion

1. Break into non-overlapping subproblems of the same type
2. Solve subproblems
3. Combine results

Linear Search: Input: an Array A with n elements. A key K. Output: An index I, where A[i] = k If there is no such I, then NOT_FOUD

Definition: A recurrence relation is an equation recursively defining a sequence of values;

```
LinearSearch(A, low, high, key)
if high < low:
  return NOT_FOUND
if A[low] = key:
  return low
return LinearSearch(A, low + 1, high, key)
```

Linear Searh Recurrence: T(n) = T(n-1) + c . T(0) = c c includes checking high verses low, checking if A[low] equals key, and preparing the parameters for recursive call

$$\sum_{i=0}^{n} c = \theta(n)$$

Binary Search: 必须是 sorted 的 array， 最长时间 search 是$(\log_2 n) + 1$

```
BinarySearchIt(A, low, high, key)
while low ≤ high:
  mid ← ⌊low + (high−low)/2⌋
  if key = A[mid]:
    return mid
  else if key < A[mid]:
    high = mid − 1
  else:
    low = mid + 1
return low − 1
```

Polynomial Multiplication

Naïve Implementation

```
MultPoly(A, B, n)
product ← Array[2n − 1]
for i from 0 to 2n − 2:
  product[i] ← 0
for i from 0 to n − 1:
  for j from 0 to n − 1:
    product[i + j] ← product[i + j] + A[i] ∗ B[j]
```

Runtime: O(n^2)

Break into several parts

- Let $A(x) = D_1(x)x^{\frac{n}{2}} + D_0(x)$ where
  $D_1(x) = a_{n-1}x^{\frac{n}{2}-1} + a_{n-2}x^{\frac{n}{2}-2} + \ldots + a_{\frac{n}{2}}$
  $D_0(x) = a_{\frac{n}{2}-1}x^{\frac{n}{2}-1} + a_{\frac{n}{2}-2}x^{\frac{n}{2}-2} + \ldots + a_0$
- Let $B(x) = E_1(x)x^{\frac{n}{2}} + E_0(x)$ where
  $E_1(x) = b_{n-1}x^{\frac{n}{2}-1} + b_{n-2}x^{\frac{n}{2}-2} + \ldots + b_{\frac{n}{2}}$
  $E_0(x) = b_{\frac{n}{2}-1}x^{\frac{n}{2}-1} + b_{\frac{n}{2}-2}x^{\frac{n}{2}-2} + \ldots + b_0$
- $AB = (D_1x^{\frac{n}{2}} + D_0)(E_1x^{\frac{n}{2}} + E_0)$
  $= (D_1E_1)x^n + (D_1E_0 + D_0E_1)x^{\frac{n}{2}} + D_0E_0$
- Calculate D_1E_1, D_1E_0, D_0E_1, and D_0E_0

Recurrence: $T(n) = 4T(\frac{n}{2}) + kn$

Karatsuba approach

$A(x) = a_1x + a_0$

$B(x) = b_1x + b_0$

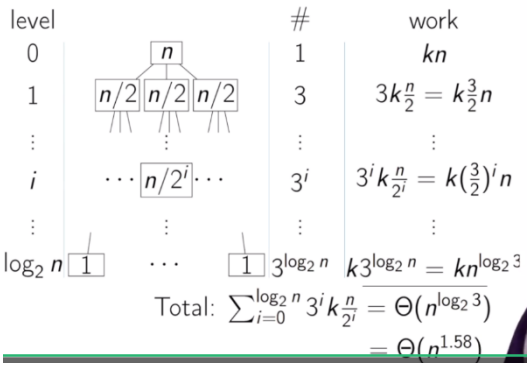
$C(x) = a_1b_1x^2 + (a_1b_0 + a_0b_1)x + a_0b_0$

Needs 4 multiplications

Rewrite as:

$C(x) = a_1b_1x^2 + ((a_1 + a_0)(b_1 + b_0) - a_1b_1 - a_0b_0)x + a_0b_0$

Needs 3 multiplications



Now running time is : O(n^1.58)

Master Theorem

If T(n) = aT(n/b) + O(n^d) (n/b 是 ceiling 不是 floor) where a> 0 and b> 1 d>=0 (b>0 的原因是想要 problem smaller)

B 是要分成几份，a 是在那一层上有几个 operation

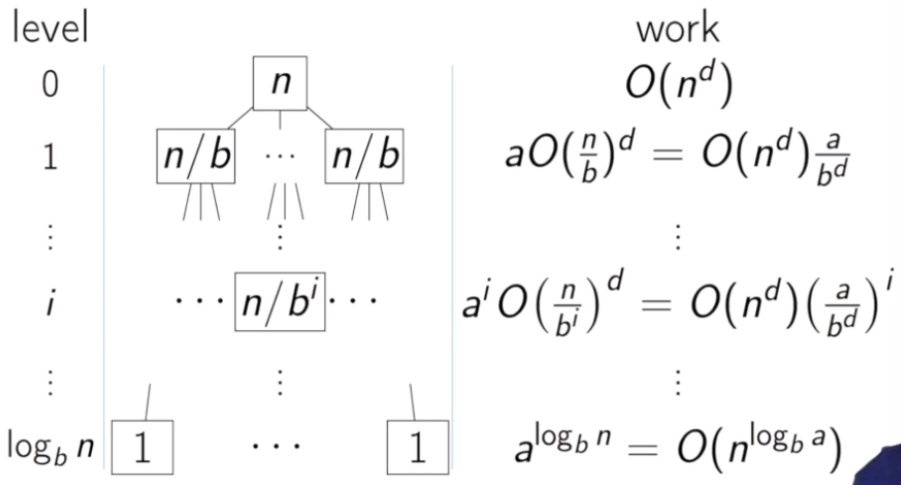
Then T(n)= $\begin{cases} O(n^d) & if\ d > \log_b a \\ O(n^d \log n) & if\ d = \log_b a \\ O(n^{\log_b a}) & if\ d < \log_b a \end{cases}$

Example 1 : naïve polynomial multiplication: T(n) = 4 T(n/2) + O(n)

A = 4 , b = 2, d = 1, d < logb(a) it is $O(n^{\log_b a}) = O(n^2)$

Example 2: efficient polynomial multiplication: T(n) = 4 T(n/3) + O(n) A = 4 , b = 3, d = 1, d < logb(a) it is $O(n^{\log_b a}) = O(n^{1.5})$

Example 3: binary search example : T(n) = T(n/2) + O(1) a = 1, b = 2 , d = 0 ,) it is $O(logn)$



Sorting Problem

Input: Sequence A[1,…, n] Ouput: permutation A'[1…n] of A[1,….,n] in non-decreasing order

Selection Sort: 1. Find a minimum by scanning the array 2. Swap it with the first element 3. Repeat with the remaining part of the array

有 list， 先找从 1-n, 最小的 number，然后 swap 1 和最小的这个 number， 然后忘记 index 1， 再从 2 – N 找最小的 number，然后 swap 2 和最小的这个 element index。 然后从第三个开始。。。。直到 n。

```
SelectionSort(A[1...n])
for i from 1 to n:
  minIndex ← i
  for j from i + 1 to n:
    if A[j] < A[minIndex]:
      minIndex ← j
  {A[minIndex] = min A[i...n]}
  swap(A[i], A[minIndex])
  {A[1...i] is in final position}
```

The running time of selection sort doesnot depend on the input of data

The running time is O(n^2) (outer loop n, inner loop n)

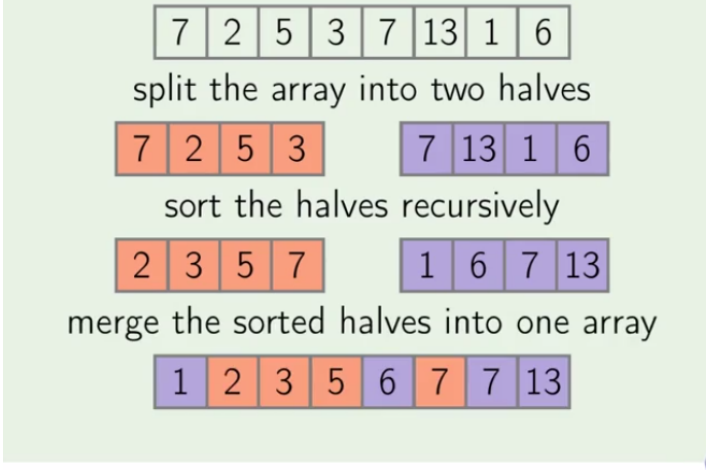
Sorts in place: requires a constant amount of extra momery.

Some other sorting algorithm with O(n^2) run time: insertion sort, bubble sort

Insertion sort: list size n, 1. 首先从第一个开始，判断第一个数和第二个数是不是 sorted order，不是的话，调换顺序。2. 判断加上第三个数 是不是 sorted array，loop 1-2 是不是第三个数都大于 1,2 的数，如果不大于的，insert 第三个数到第一或者第二位置。,3. 判断加上第四个数是不是 sorted order，for loop 1-3 插入第四个数到合适的位置。。。。。直到 n 结束

Merge Sort:

Example: merge sort




```
MergeSort(A[1...n])
if n = 1:
  return A
m ← ⌊n/2⌋
B ← MergeSort(A[1...m])
C ← MergeSort(A[m+1...n])
A' ← Merge(B, C)
return A'
```

```
Merge(B[1...p], C[1...q])
{B and C are sorted}
D ← empty array of size p+q
while B and C are both non-empty:
  b ← the first element of B
  c ← the first element of C
  if b ≤ c:
    move b from B to the end of D
  else:
    move c from C to the end of D
move the rest of B and C to the end of D
return D
```

Running time of Merge 是 p + q

Total running time of merge sort is O(nlogn)

proof

The running time of merge B and C is O(n) 因为 b,c 都是 sorted 的，所以 b 是 n/2, C 是 n/2, 所以结合 b，c 就是 O(n)

Hence the running time of Mergesort statisfies a recurrence T(n) < 2T(n/2) + O(n)

O(nlogn) log n 是 log n 部 split combine 的步数，n 是结合每一步，结合 B,C 的数

No sorting algoritm can perform better than O(nlogn)

Lemma: Any comparison based sorting algorithm performs $\Omega(\text{nlogn})$ comparisons in the worst case to sort n objects

In other words comparison based sorting algorithm perform at least $\Omega(\text{nlogn})$ comparisons to sort A.

Proof Decision tree 的 leaves l 的个数 at least n！

The worst case running time of the algorithm is at least the depth of d where $d \geq \log_2 l$

The running time is at least $\log(n!) = \Omega(nlogn)$

$$\log(n!) = log2(1 * 2 * \ldots * n)$$

$$= \log_2 1 + \log_2 2 + \log_2 3 + \ldots + \log_2 n$$

$$\geq \log_2 \frac{n}{2} + \cdots . + \log_2 n$$

$$\geq \frac{n}{2} \log_2 \frac{n}{2} = \Omega(nlogn)$$

Non-comparative sorting：

给你一个 list， 然后 list 只有 integer from 1 to M, 只要数每一个数在这 list 出现的次数就可以了， without compare, running time O(n+M) n 是 list 的 size， M 是有多少种数在 list 中

Quick sort

1. Comparison based algorithm
2. Running time: O(nlogn) (on average)

Example: quick sort

6	4	8	2	9	3	9	4	7	6	1

partition with respect to $x = A[1]$

in particular, x is in its final position

1	4	2	3	4	6	6	9	7	8	9

sort the two parts recursively

1	2	3	4	4	6	6	7	8	9	9

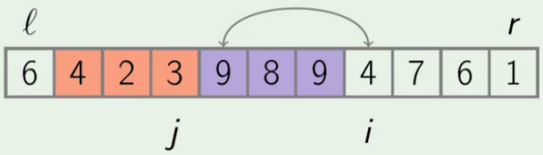
QuickSort(A, ℓ, r)

```
if ℓ ≥ r:
  return
m ← Partition(A, ℓ, r)
{A[m] is in the final position}
QuickSort(A, ℓ, m − 1)
QuickSort(A, m + 1, r)
```

Pivot Element：the element we are going to use to partition our sub-array . x will be placed at the final position/

Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell + 1$ to r maintaining the following invariant:
  - $A[k] \le x$ for all $\ell + 1 \le k \le j$
  - $A[k] > x$ for all $j + 1 \le k \le i$



要保证 pivot 的左面的数都要小于 pivot，右边的数都要大于 pivot，所以在 swap pivot 和 final position 前，就 swap 其他的数，保证左面数小，右面数大

Partition(A, ℓ, r)

```
x ← A[ℓ]    {pivot}
j ← ℓ
for i from ℓ + 1 to r:
  if A[i] ≤ x:
    j ← j + 1
    swap A[j] and A[i]
  {A[ℓ + 1 . . . j] ≤ x, A[j + 1 . . . i] > x}
swap A[ℓ] and A[j]
return j
```

UnBalancd Partitations: T(n) = n + T(n-1) = n + (n-1) + (n-2) + ... = $\Omega(n^2)$

The running time is depended on how balanced our partition

Unbalanced runtime is O(n^2) . Balanced runtime is O(nlogn)

Quick sort 最糟糕的情况是 每个数都比一次，

```
RandomizedQuickSort(A, ℓ, r)
if ℓ ≥ r:
  return
k ← random number between ℓ and r
swap A[ℓ] and A[k]
m ← Partition(A, ℓ, r)
{A[m] is in the final position}
RandomizedQuickSort(A, ℓ, m − 1)
RandomizedQuickSort(A, m + 1, r)
```

Proof：

Proof (continued)

Then (the expected value of) the running time is

$$\begin{aligned} \mathrm{E}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\chi_{ij} &= \sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}(\chi_{ij}) \\ &= \sum_{i<j}\frac{2}{j-i+1} \\ &\leq 2n\cdot\left(\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}\right) \\ &= \Theta(n\log n) \end{aligned}$$

Xij 是 probability 两个数比的概率，假如 array 有 1 到 9，9 个数，两个数离得很远 1,9 被比较的概率是 2/9 因为假如 pivot 是 3，这样的话 1 和 9 被分成两组，不会被比较，只有当 pivot 选取 1 或者 9，两个数才会被比较，2/(9-1+1) = 2/9

但是当两个数离得很近，3,4 被比的概率是 1，因为最后的 pivot 肯定是 3,4 中选一个 2/(4-3+1) = 1

Quick sort work quadratic time when elements are equal。 Quick sort 不 work well 当元素都差不多相同的时候

To address this problem, split into three regions

To handle equal elements, we replace the line

$$m \leftarrow \texttt{Partition}(A, \ell, r)$$

with the line

$$(m_1, m_2) \leftarrow \texttt{Partition3}(A, \ell, r)$$

such that

- for all $\ell \leq k \leq m_1 - 1$, $A[k] < x$
- for all $m_1 \leq k \leq m_2$, $A[k] = x$
- for all $m_1 + 1 \leq k \leq r$, $A[k] > x$

```
RandomizedQuickSort(A, ℓ, r)
if ℓ ≥ r:
  return
k ← random number between ℓ and r
swap A[ℓ] and A[k]
(m1, m2) ← Partition3(A, ℓ, r)
{A[m1 ... m2] is in final position}
RandomizedQuickSort(A, ℓ, m1 − 1)
RandomizedQuickSort(A, m2 + 1, r)
```

```
QuickSort(A, ℓ, r)
while ℓ < r:
  m ← Partition(A, ℓ, r)
  if (m − ℓ) < (r − m):
    QuickSort(A, ℓ, m − 1)
    ℓ ← m + 1
  else:
    QuickSort(A, m + 1, r)
    r ← m − 1
```

减少 recursion 的次数

Week 5 Dynamic Programming

Dynamic programming ：最终点的状态取决前一个，苏北 problem 前一个状态的 optimal solution

How much cents to be changed for 40? (US = 25 + 9 + 1, 坦桑尼亚 40 = 20 + 20

建一个 list，从 1-n， 每一步都是最 optimal，loop from 1 to n， 然后 n 就是答案

```
DPChange(money, coins)
MinNumCoins(0) ← 0
for m from 1 to money:
  MinNumCoins(m) ← ∞
  for i from 1 to |coins|:
    if m ≥ coin_i:
      NumCoins ← MinNumCoins(m − coin_i) + 1
      if NumCoins < MinNumCoins(m):
        MinNumCoins(m) ← NumCoins
return MinNumCoins(money)
```

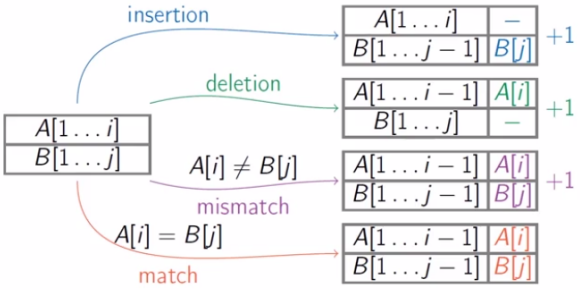
String Comparison

Edit Distance: Input: Two strings

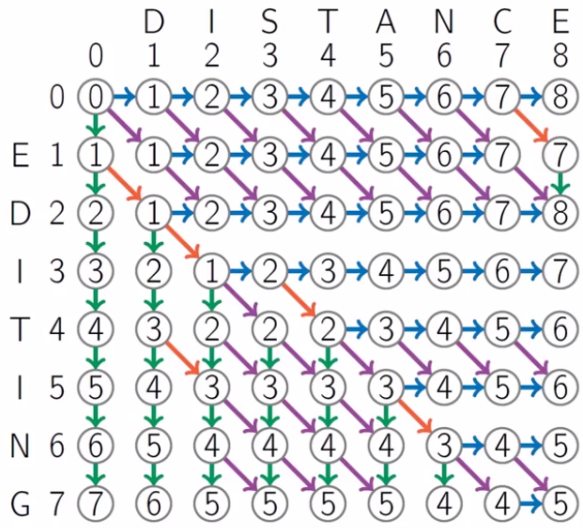
Output: The minimum， ， number of operations (insertions, deletions, and substitutions of symbol) to transform one string into another.

最小的操作把一个 string 转换成另一个 string

Minimizing edit distance = maximizing alignment score.



$$D(i,j) = \min \begin{cases} D(i,j-1)+1 & \\ D(i-1,j)+1 & \\ D(i-1,j-1)+1 & \text{if } A[i] \neq B[j] \\ D(i-1,j-1) & \text{if } A[i] = B[j] \end{cases}$$



Filling the dynamic programming matrix

```
EditDistance(A[1...n], B[1...m])
D(i,0) ← i and D(0,j) ← j for all i,j
for j from 1 to m:
  for i from 1 to n:
    insertion ← D(i,j−1) + 1
    deletion ← D(i−1,j) + 1
    match ← D(i−1,j−1)
    mismatch ← D(i−1,j−1) + 1
    if A[i] = B[j]:
      D(i,j) ← min(insertion, deletion, match)
    else:
      D(i,j) ← min(insertion, deletion, mismatch)
return D(n,m)
```

Knapsack

With repetition

Let $value(w)$ be the maximum value of knapsack of weight w.

$$value(w) = \max_{i:\, w_i \leq w} \{value(w - w_i) + v_i\}$$

```
Knapsack(W)
value(0) ← 0
for w from 1 to W:
  value(w) ← 0
  for i from 1 to n:
    if w_i ≤ w:
      val ← value(w − w_i) + v_i
      if val > value(w):
        value(w) ← val
return value(W)
```

Running time O(wn)

Without repetition

Subproblems

For $0 \le w \le W$ and $0 \le i \le n$, $value(w, i)$ is the maximum value achievable using a knapsack of weight w and items $1, \ldots, i$.

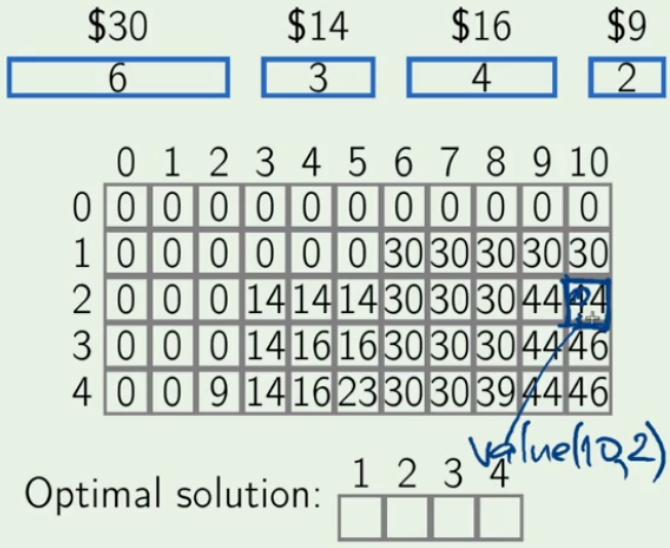
The i-th item is either used or not: $value(w, i)$ is equal to

$$\max\{value(w-w_i, i-1)+v_i, value(w, i-1)\}$$

Knapsack(W)

```
initialize all value(0, j) ← 0
initialize all value(w, 0) ← 0
for i from 1 to n:
  for w from 1 to W:
    value(w, i) ← value(w, i − 1)
    if w_i ≤ w:
      val ← value(w − w_i, i − 1) + v_i
      if value(w, i) < val
        value(w, i) ← val
return value(W, n)
```

Example: reconstructing a solution



Value(10,2)的意思是在 weight 等于 10 时，只有前两个 item 被使用， value(10-3, 1) + v2， 在 weight 为 10，value v2 被使用，value(10-3, 1) value 7, value v1 被使用

Value (10,2) = max(value(10-3, 1) + v2, value(10, 1))

Memoization：记录之前的状态，avoid re-computing many times the same thing. 比如 Fibonacci number 的 recursion

If all subproblems must be solved then an iterative algorithm is usually fster since it has no recursion overhead

Running Time

- The running time $O(nW)$ is not polynomial since the input size is proportional to $\log W$, but not W.
- In other words, the running time is $O(n2^{\log W})$.
- E.g., for

  $W = 71\,345\,970\,345\,617\,824\,751$

  (twenty digits only!) the algorithm needs roughly 10^{20} basic operations.

Placing Paraentheses

给数字然后给加减乘除，place 括号给最大值

Subproblems

Ei.j the optimal value from i to j

- Let $E_{i,j}$ be the subexpression

$$d_i\, op_i \cdots op_{j-1}\, d_j$$

- Subproblems:

$$M(i,j) = \text{maximum value of } E_{i,j}$$
$$m(i,j) = \text{minimum value of } E_{i,j}$$

MinAndMax(i,j)

$min \leftarrow +\infty$
$max \leftarrow -\infty$
for k from i to $j-1$:
  $a \leftarrow M(i,k) \quad op_k \quad M(k+1,j)$
  $b \leftarrow M(i,k) \quad op_k \quad m(k+1,j)$
  $c \leftarrow m(i,k) \quad op_k \quad M(k+1,j)$
  $d \leftarrow m(i,k) \quad op_k \quad m(k+1,j)$
  $min \leftarrow \min(min, a, b, c, d)$
  $max \leftarrow \max(max, a, b, c, d)$
return (min, max)

1. when computing M(I,j), the values of M(i, k) and M(k+1, j) should be already computed.
2. M(i,j) 代表着 值取从 I 开始 到 j 结束的值

Parentheses$(d_1\, op_1\, d_2\, op_2 \ldots d_n)$

for i from 1 to n:
  $m(i,i) \leftarrow d_i$, $M(i,i) \leftarrow d_i$
for s from 1 to $n-1$:
  for i from 1 to $n-s$:
    $j \leftarrow i+s$
    $m(i,j), M(i,j) \leftarrow$ MinAndMax(i,j)
return $M(1,n)$

initialize max and min value table ,中间的 digonal 代表，只取那一个数的值，就是那个数

Example: $5 - 8 + 7 \times 4 - 8 + 9$

(1,2) (3,3)

	1	2	3	4	5	6
1	5	-3	-10	-55	-63	-94
2		8	15	36	-60	-195
3			7	28	-28	-91
4				4	-4	-13
5					8	17
6						9

m

	1	2	3	4	5	6
1	5	-3	4	25	65	200
2		8	15	60	52	75
3			7	28	20	35
4				4	-4	5
5					8	17
6						9

M

Course 2 Data Structure

Week 1 Array / Linked List / Stack / Queue / Trees

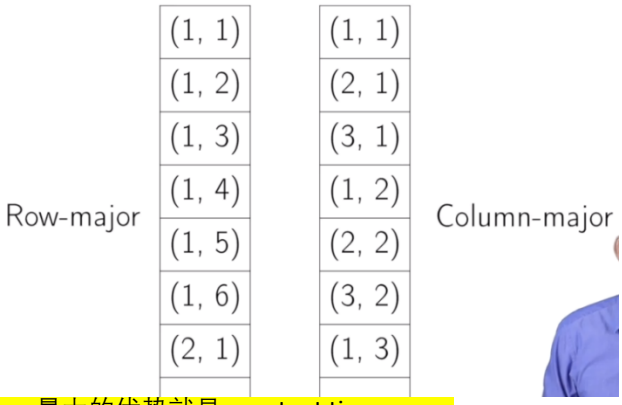
Array / Linked List

Definition Array : Contiguous area of memory consisting of equal-size elements indexed by contiguous integers;

Key points of array: random access (constant time to read / write)

Array_addr + element_size X (i- first_index)

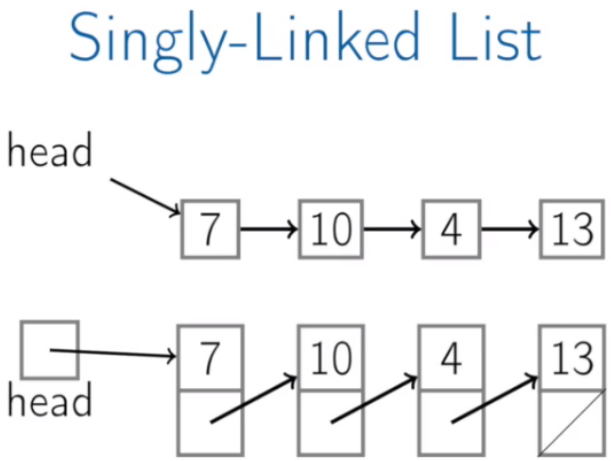
Multi-dimensional array : ordering , 分为 column major， row major



	Add	Remove
Beginning	$O(n)$	$O(n)$
End	$O(1)$	$O(1)$
Middle	$O(n)$	$O(n)$

Array 最大的优势就是 constant time access

Linked List: Each nodes contains key and next pointer



Operation can be with Linked List:

List API	
PushFront(Key)	add to front
Key TopFront()	return front item
PopFront()	remove front item
PushBack(Key)	add to back
Key TopBack()	return back item
PopBack()	remove back item
Boolean Find(Key)	is key in list?
Erase(Key)	remove key from list
Boolean Empty()	empty list?
AddBefore(Node, Key)	adds key before node
AddAfter(Node, Key)	adds key after node

Singly-Linked List	no tail	with tail
PushFront(Key)	$O(1)$	
TopFront()	$O(1)$	
PopFront()	$O(1)$	
PushBack(Key)	$O(n)$	$O(1)$
TopBack()	$O(n)$	$O(1)$
PopBack()	$O(n)$	
Find(Key)	$O(n)$	
Erase(Key)	$O(n)$	
Empty()	$O(1)$	
AddBefore(Node, Key)	$O(n)$	
AddAfter(Node, Key)	$O(1)$	

Push Front: O(1) Pop Front : O(1)

PushBack(no tail) O(n) Pop Back(no tail) O(n)

PushBack(with tail) O(1) Pop Back(with tail) O(n) 因为即使有最后的 pointer，但不能从倒数第一个到倒数二个，单向的

Doubly-Linked List:

Make addbefore and pop_back O(1)

1. O(n) time to find arbitrary element
2. With Doubly – Linked list, const time to insert between nodes or remove a node
3. List elements need not be contiguous, (array 需要 element contiguous)

Stack / Queues

Definition : Stack: abstract data type with following operations: LIFO

1. Push (key): adds key to collection
2. Key Top(): returns most recently-added key
3. Key Pop: removes and returns most recently-added key
4. Boolean empty(): are there any elements?

可以 add / remove key on the top, 但是不能 add / remove key at the bottom

Balanced Brackets

Balanced:

- ‘‘([])[]()’’,
- ‘‘((([([])]))())’’

Unbalanced:

- ‘‘([]]()’’
- ‘‘][’’

IsBalanced(*str*)

```
Stack stack
for char in str:
  if char in ['(', '[']:
    stack.Push(char)
  else:
    if stack.Empty():  return False
    top ← stack.Pop()
    if (top = '[' and char != ']') or
      (top = '(' and char != ')'):
      return False
return stack.Empty()
```

Implement stack with Linked List, One disadvantage of Array is that it has fixed size

Each stack operation is O(1), Push, Pop， Top, Empty

Definition : Queue: abstract data type with following operations: FIFO

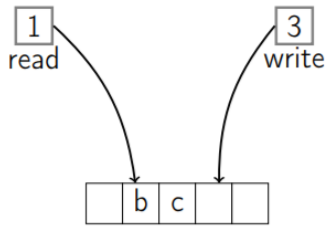
1. Enqueue (key): adds key to collection
2. Key Dequeue(): removes and returns least recently-added key
3. Boolean empty(): are there any elements?

Implement Queue with Linked List:

1. Enqueue: use list PushBack
2. Dequeue: use list. Top front and list.Popfront
3. Empty: use List Empty

Queue Implemention with Array： circular, 保证 push pop O(1), left at least position as buffer 去识别头和尾，一旦头和尾不重合，就不能区别了

Queue Implementation with Array



Dequeue() → a

Each Queue Operation is O(1), Enqueue, Dequeue, Empty

Tree

Binary Search Tree: It means it has at most two children at each node. The root node 都大于 left child, root node value 都小于 right child. It also applies for every nodes of the tree

Termiology：

Root： top node in the tree

Child: A child has a line down directly from a parent

Ancestor: parent or parent of parent, etc

Descendant: child, child of child

Sibling: sharing the same parents

Leaf: node with no children

Interior node: non-leave

Level: 1+ num edges between root and node

Height: maximum depth of subtree node and farthest leaf

Forest: collection of trees

Node contains: 1. Key 2. Children : list of children nodes. 3. (optional_ parent

For binary tree: node contains; 1. Key 2. Left children 3. Right children 3. (optional) parents

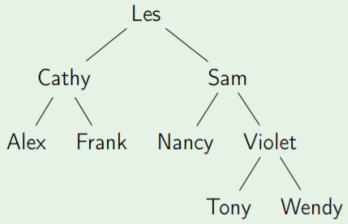
Tree Traversal

1. Depth First: we completely traverse one sub-tree before exploring a sibling sub-tree (DFS)
2. Breadth-first: We traverse all nodes at one level before progressing to the next level (BFS)

Depth-first

```
InOrderTraversal(tree)
if tree = nil:
  return
InOrderTraversal(tree.left)
Print(tree.key)
InOrderTraversal(tree.right)
```

InOrderTraversal

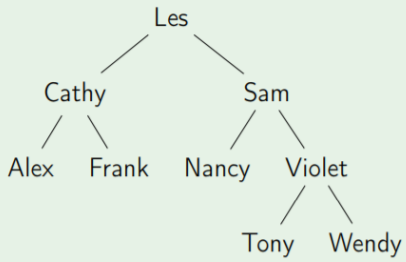


Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

Depth-first

```
PreOrderTraversal(tree)
if tree = nil:
  return
Print(tree.key)
PreOrderTraversal(tree.left)
PreOrderTraversal(tree.right)
```

PreOrderTraversal

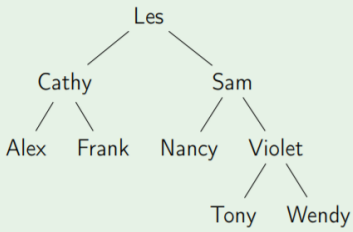


Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

Depth-first

```
PostOrderTraversal(tree)
if tree = nil:
  return
PostOrderTraversal(tree.left)
PostOrderTraversal(tree.right)
Print(tree.key)
```

PostOrderTraversal



Output: Alex Frank Cathy Nancy Tony Wendy Violet Sam Les

In Pre Post 定义根据第几个读 node

InOrder: 左 中 友

Preorder: 中 左 右

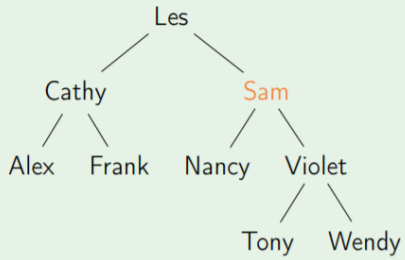
Post Order 左 右中

Recursion 的时候 invoke another frame on the stack, saving implicitly our information of we are on the stack

Breadth-first

```
LevelTraversal(tree)
if tree = nil:  return
Queue q
q.Enqueue(tree)
while not q.Empty():
  node ← q.Dequeue()
  Print(node)
  if node.left ≠ nil:
    q.Enqueue(node.left)
  if node.right ≠ nil:
    q.Enqueue(node.right)
```

LevelTraversal



Output: Les Cathy Sam

Queue: Alex, Frank, Nancy, Violet

Week 2 Dynamic Arrays and Amortized Analysis

Dynamic Array

Problem of array: static, size does not change

Semi-Solution: dynamically-allocate arrays;

Int *my_array = new int [size] // determine the size of array at the runtime

Problem: 也许不知道 max size when allocating an array

Solution: dynamic arrays (also known as resizable arrays)

Idea: store a pointer to a dynamically allocated array , and replace it with a newly-allocated array as needed

Definition: Abstract data type with the following (at a minimum):

1. Get(i): returns element at location I O(1)
2. Set(I, val): sets element I to val* O(1)
3. PushBack(val)L Adds val to the end
4. Remove(i): Removes element at location i
5. Size(): the number of elements

Implementation: Store:

1. Arr: dynamically-allocated array
2. Capacity : size of the dynamically – allocated array
3. Size: number of elements currently in the array

PushBack(*val*)

```
if size = capacity:
  allocate new_arr[2 × capacity]
  for i from 0 to size − 1:
    new_arr[i] ← arr[i]
  free arr
  arr ← new_arr; capacity ← 2 × capacity
arr[size] ← val
size ← size + 1
```

Dynamic array Resizing: 当 size 超过 capacity, resize

C++ : vector

Java: ArrayList

Python: list (the only kind of array)

Runtimes

Get(*i*)	$O(1)$
Set(*i*, *val*)	$O(1)$
PushBack(*val*)	$O(n)$
Remove(*i*)	$O(n)$
Size()	$O(1)$

Summary:

1. Unlike static arrays, dynamic arrays can be resized
2. Appending a new element to a dynamic array is often constant time but can time O(n)
3. Some space is wasted

Amortized Analysis:

Amortized cost: given a sequence of n operations, the amortized cost is $\frac{\text{Cost(n operations)}}{\text{n}}$

Aggregate Method: n calls to PushBack, Let $C_i = cost\ of\ i'th\ \ insertion$

$$C_i = 1 + \begin{cases} i-1, if\ i-1\ is\ a\ pwoer\ of\ 2 \\ 0\ \ otherwise \end{cases}$$

$$\frac{\sum_{i=1}^{n} C_i}{n} = \frac{n + \sum_{j=1}^{\log_2(n-1)|} 2^j}{n} = \frac{O(2n)}{n} = O(1)$$

可以是 2 的倍数的 resize，但是不能使 constant amount 比如 10 的倍数的 resize （10,100， 1000， ），这样的话就是 O(n) operations, 用 constant factor 不是 constant amount

---

Cannot Use Constant Amount

If we expand by 10 each time, then:

Let c_i = cost of i'th insertion.

$$c_i = 1 + \begin{cases} i-1 & \text{if } i-1 \text{ is a multiple of 10} \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^{n} c_i}{n} = \frac{n + \sum_{j=1}^{(n-1)/10} 10j}{n} = \frac{n + 10\sum_{j=1}^{(n-1)/10} j}{n}$$

$$= \frac{n + 10O(n^2)}{n} = \frac{O(n^2)}{n} = O(n)$$

Week 3 Priority Queue And Disjoint Sets

Priority Queues

C++: priority_queue Java: PriorityQueue Python: heapq

A priority queue is a generalization of a queue where each element is assigned a priority and elements come out in order by priority

Definition: priority queue is an abstract data type supporting the following main operations:

1. Insert(p): adds a new element with priority p
2. ExtractMax(): extracts an element with maximum priority

Additional Operations:

1. Remove(it) removes an element pointed by an iterator it
2. GetMax() returns an element with maximum priority (without changing the set of elements)
3. ChangePriority(it,p) changes the priority of element pointed by it to p

Naïve Implementations:

1. 用 array 或者 linked list,
   Insert(e) add e to the end, runtime O(1)
   ExtractMax(): sacen the array/list running time O(n)

2. Sorted Array
   ExtractMax(): extract the last element (which is the biggest) runtime O(1)
   Insert(e) ： find a position for e (O(logn)) by using binary search), shift all elements to the right of it by 1 O(n), insert e(O(1)), total running time O(n)

3. Sorted List
   ExtractMax(): runtime O(1)
   Insert(e) ： find a position for e O(n) (不能用 binary search), insert e(O(1)), total running time O(n)

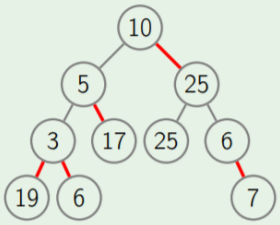
Summary

	Insert	ExtractMax
Unsorted array/list	$O(1)$	$O(n)$
Sorted array/list	$O(n)$	$O(1)$
Binary heap	$O(\log n)$	$O(\log n)$

Priority Queues： Heaps

Definition: Binary max-heap is a binary tree (each node has zero, one, or two children) where the value of each node is at least the values of its children. （parent 的数大于 children 的数）

Example: not a heap



Definition of height of heap: the number of ***edges*** on the longest path from the root to a leaf. 上面就是 height = 3

The advantage of binary heap is to find the maximum value without extracting it. (return root value)

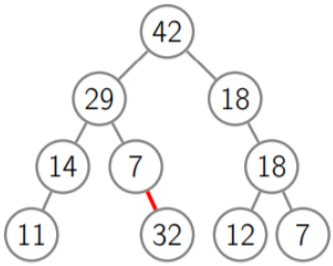
Insertion: attach a new node （value） to any leaf， then fix the rule, sift up the new elements

SiftUp

Invariant: heap property is violated on at most one edge .

Running time O(logn)

for this, we swap the problematic node with its parent until the property is satisfied



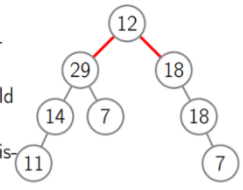
ExtractMax: 先 extract， 然后 replace the root with any leaf, 之后与选左和右 child 的最大值，跟 root swap

SiftDown

（sift down）

Runing time O(logn)

for this, we swap the problematic node with larger child until the heap property is satisfied



Change Priority: running time O(logn) 把其中一个元素的 priority 变大或者变小， sift up/ sift down

Remove: change the priorty of the element to infinity, sift up, then ExtractMax, running time O(logn)

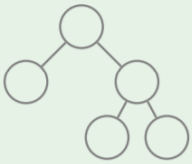
Summary: 1. getMax works in O(1), all other operations work in time O(tree height)

2 . We definitely want a tree to be shallow: (keep nodes all packed in every layers)

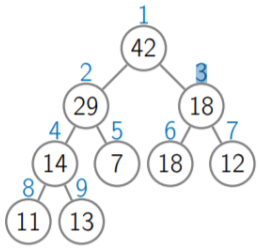
Definition: A binary tree is complete if all its levels are filled except possibly the last one which is filled from left to right (left most position)

A complete binary tree with n nodes has height at most O(logn)

Example: not complete binary tree



Second Advantage: Store as Array



$\text{parent}(i) = \lfloor \frac{i}{2} \rfloor$

$\text{leftchild}(i) = 2i$

$\text{rightchild}(i) = 2i + 1$

1	2	3	4	5	6	7	8	9
42	29	18	14	7	18	12	11	13

Keep the Tree Complete:

Insertion: To insert an element, insert it as a leaf in the leftmost vacant position in the last level and let it sift up.

ExtractMax: to extract the maximum value replace the root by the last leaf and let it sift down

SiftUp(i)

```
while i > 1 and H[Parent(i)] < H[i]:
  swap H[Parent(i)] and H[i]
  i ← Parent(i)
```

Insert(p)

```
if size = maxSize:
  return ERROR
size ← size + 1
H[size] ← p
SiftUp(size)
```

SiftDown(i)

```
maxIndex ← i
ℓ ← LeftChild(i)
if ℓ ≤ size and H[ℓ] > H[maxIndex]:
  maxIndex ← ℓ
r ← RightChild(i)
if r ≤ size and H[r] > H[maxIndex]:
  maxIndex ← r
if i ≠ maxIndex:
  swap H[i] and H[maxIndex]
  SiftDown(maxIndex)
```

ExtractMax()

```
result ← H[1]
H[1] ← H[size]
size ← size − 1
SiftDown(1)
return result
```

Remove(i)

```
H[i] ← ∞
SiftUp(i)
ExtractMax()
```

ChangePriority(i, p)

```
oldp ← H[i]
H[i] ← p
if p > oldp:
  SiftUp(i)
else:
  SiftDown(i)
```

Summary: The resulting implementation is :

1. Fast: all operations work in time O(log n) (GetMax even work in O(1))
2. Space efficient: we store an array of priorities, parent-child connections are not stored, but are computed on the fly
3. Easy to implement, All operations are implemented in just few lines of code

Priority Queues： Heap sort

Heap

HeapSort($A[1 \dots n]$)

```
create an empty priority queue
for i from 1 to n:
  Insert(A[i])
for i from n downto 1:
  A[i] ← ExtractMax()
```

1. The result algorithm is comparion – based and has running time O(nlogn)
2. Natural generalization of selection sort; instead of simply scanning the rest of the array to fnd the maximum value, use a smart data structure
3. Not in-place: use additional space to store the priority queue

BuildHeap($A[1 \dots n]$)

```
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

We turn array into a heap, repair heap property going from bottom to top

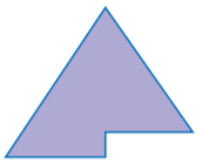
HeapSort($A[1 \dots n]$)

```
BuildHeap(A)                    {size = n}
repeat (n − 1) times:
  swap A[1] and A[size]
  size ← size − 1
  SiftDown(1)
```

In place Heap sort advantage: 1. Worst running time O(nlogn) 2. No additional space, no extra space

In practice use quick sort, 如果 quick sort 速度低于(recursion steps 超过) clogn, 设一个 constant c，switch to heap sort

Building Running Time



# nodes	T(SiftDown)
1	$\log_2 n$
2	
⋮	⋮
$\leq n/4$	2
$\leq n/2$	1

$$T(\texttt{BuildHeap}) \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \dots$$
$$\leq n \cdot \sum_{i=1}^{\infty} \frac{i}{2^i} = 2n$$

Partial sorting

Input: An array $A[1 \dots n]$, an integer $1 \leq k \leq n$.

Output: The last k elements of a sorted version of A.

Can be solved in $O(n)$ if $k = O(\frac{n}{\log n})$!

如果不 sort 全部的，可以在 O(n) 时间内完成

Building heap 最多需要 2n 的时间，但是不是 sort array

PartialSorting($A[1 \ldots n], k$)

```
BuildHeap(A)
for i from 1 to k:
  ExtractMax()
```

Running time: $O(n + k \log n)$

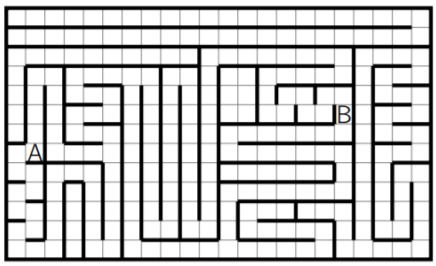
如果 k = O(n/logn) , sort 是 O（）

Binary min – heap is a binary tree where the value of each node is at most the values of its children (与 binary -max heap 的顺序是相反的)

D-ary heap: In a d-ary heap nodes on all levels except for possibly the last one have exactly d children 每一个 node 都有 d 个 children， The height of such a tree is about $\log_d n$. The running time of Siftup is O($\log_d n$), The running time of siftDown is O($d\log_d n$) , on each level, we children.

Disjoint Set

Maze: Is B Reachable from A?



Definition

A disjoint-set data structure supports the following operations:

- MakeSet(x) creates a singleton set $\{x\}$
- Find(x) returns ID of the set containing x:
  - if x and y lie in the same set, then Find(x) = Find(y)
  - otherwise, Find(x) ≠ Find(y)
- Union(x, y) merges two sets containing x and y

Preprocess(*maze*)

```
for each cell c in maze:
  MakeSet(c)
for each cell c in maze:
  for each neighbor n of c:
    Union(c, n)
```

IsReachable(A, B)

```
return Find(A) = Find(B)
```

Use the smallest element of a set as its ID

Example

$\{9, 3, 2, 4, 7\}$ $\{5\}$ $\{6, 1, 8\}$

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

MakeSet(i)

```
smallest[i] ← i
```

Find(i)

```
return smallest[i]
```

Running time: $O(1)$

Union(i, j)

```
i_id ← Find(i)
j_id ← Find(j)
if i_id = j_id:
  return
m ← min(i_id, j_id)
for k from 1 to n:
  if smallest[k] in {i_id, j_id}:
    smallest[k] ← m
```

Running time of Union (I,j) O(n)

Use Linked List:

Pro : 1. Running time of Union is O(1) 2. Well -defined ID if two elements are from the same set. Then tail is the same

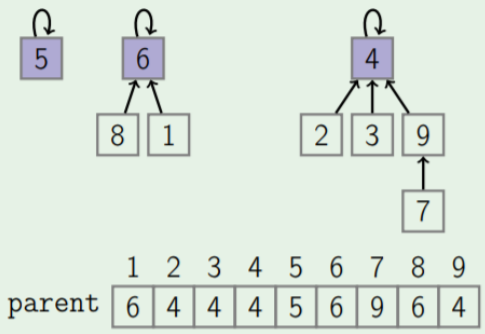
Con: 1. Running time of Find is O(n) as we need to traverse the list to find its tail

2 . Union(x,y) works in time O(1) only if we can get the tail of the list of x and the head of the list of y in constant time

Disjoint Set: Efficient Implementation

1. Represent each set as rooted tree
2. Id of a set is the root of the tree
3. Use array parent[1… n] : parent [i] is the parent of I, or I if it is the root

Example



MakeSet(*i*)

```
parent[i] ← i
```

Running time: $O(1)$

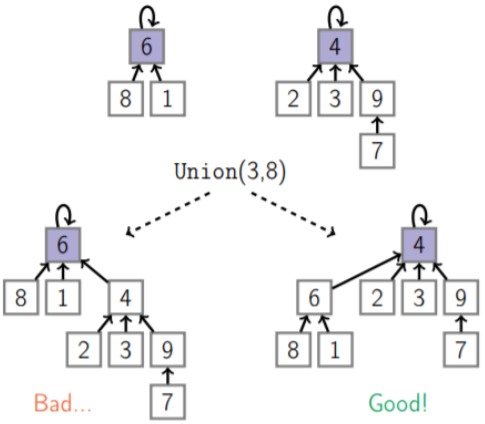
Find(*i*)

```
while i ≠ parent[i]:
  i ← parent[i]
return i
```

Running time: O(tree height)

Merge: 更改 root， hang 到另一个 tree 的 root 上，left or right subtree? Which one to hang? 谁作为 subtree，谁做为主 tree？

Answer: A shorter one, since we would like to keep the trees shallow.



To quickly find a height of each subtree in an array rank[i…n]: rank[i] is the height of the subtree whose root is i

Hanging a shorter tree under a taller one is called a union by rank heuristic

MakeSet(*i*)

```
parent[i] ← i
rank[i] ← 0
```

Find(*i*)

```
while i ≠ parent[i]:
  i ← parent[i]
return i
```

Union(*i*, *j*)

```
i_id ← Find(i)
j_id ← Find(j)
if i_id = j_id:
  return
if rank[i_id] > rank[j_id]:
  parent[j_id] ← i_id
else:
  parent[i_id] ← j_id
  if rank[i_id] = rank[j_id]:
    rank[j_id] ← rank[j_id] + 1
```

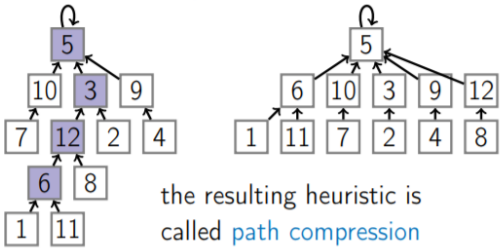
Important Property: for any node i, rank[i] is equal to the height of the tree rooted at i

The height of any tree in the forest is at most log2(n)

Summary: Th union by rank heuristic guarantees that Union and Find work in time O(logn)

Path Compression

Path Compression: Intuition



the resulting heuristic is called path compression

找 6 的 parent

Definition

The iterated logarithm of n, $\log^* n$, is the number of times the logarithm function needs to be applied to n before the result is less or equal than 1:

$$\log^* n = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$

Example

n	$\log^* n$
$n = 1$	0
$n = 2$	1
$n \in \{3, 4\}$	2
$n \in \{5, 6, \ldots, 16\}$	3
$n \in \{17, \ldots, 65536\}$	4
$n \in \{65537, \ldots, 2^{65536}\}$	5

Log*n at most 5

Lemma

Assume that initially the data structure is empty. We make a sequence of m operations including n calls to `MakeSet`. Then the total running time is $O(m \log^* n)$.

In other words

The amortized time of a single operation is $O(\log^* n)$.

When both union by rank heuristic and path compression heuristic are used, the average running time of each operation is nearly constant.

When using path compression, rank[i] is no longer equal to the height of the subtree rooted at i, height is at most the rank[i], height <= rank[i]

It is still true that a root node of rank k has at least 2**k nodes in the subtree, a root node is not affected by path compression

Week 4 Hash Tables

Introduction, Direct Addressing and Chaining

Direct Addressing requires too much memory, 如果存每个数作为 array，index，特别费 memory，需要 int【max num】作为 array

Hash Function: For any set of objects S and any integer m>0, a function h: S-> {0,1,….m-1} is called a hash function , (Hash Function 是算 chain 的起始点)

M is called cardinality.

Property: 1. H is fast to compute. 2. Different values for different objects 3. Direct addressing with O(m) memory. 4. Want a small cardinality m 5. Impossible to have all different values if number of objects |S| is more than m

Collisions: when h(O1) = h(o2) and o1!= o2 this is collision

Definition:

Map from S to V is a data structure with methods HasKey(O), Get(O), Set(O,v) where O ∈ $S, v \in V$

$h : S \to \{0, 1, \ldots, m-1\}$
$O, O' \in S$
$v, v' \in V$
$A \leftarrow$ array of m lists (chains) of pairs (O, v)

```
HasKey(O)
L ← A[h(O)]
for (O', v') in L:
  if O' == O:
    return true
return false
```

```
Get(O)
L ← A[h(O)]
for (O', v') in L:
  if O' == O:
    return v'
return n/a
```

```
Set(O, v)
L ← A[h(O)]
for p in L:
  if p.O == O:
    p.v ← v
    return
L.Append(O, v)
```

先用 hash function 算值得 hash 值，然后 loop through list，

Lemma: Let c be the length of the longest chain in A Then the running time of Hashkey, Get, Set is $\Theta(c+1)$ (因为 hash function 得到 chain 的标签是 O(1),然后 loop through 那个 chain 是 O(c)

Lemma: Let n be the number of different keys O currently in the map and m be the cardinality of the hash function, Then the memory consumption for chaining is $\Theta(n+m)$，(对于 m 来说，即使有些位置是空的，也有 store the pointers to the heads of those lists)

Definition: Set is a data structure with methods Add(O)， Remove(O), Find(O)

Two ways to implement a set using chaining

1. Set is equivalent to map from S to V={true, false}, (waste of memory)
2. Store just objects O instead of pairs (O,v) in chains

$h : S \to \{0, 1, \ldots, m-1\}$
$O, O' \in S$
$A \leftarrow$ array of m lists (chains) of objects O

Find(O)

```
L ← A[h(O)]
for O' in L:
  if O' == O:
    return true
return false
```

Add(O)

```
L ← A[h(O)]
for O' in L:
  if O' == O:
    return
L.Append(O)
```

Remove(O)

```
if not Find(O):
  return
L ← A[h(O)]
L.Erase(O)
```

Definition:

Hash Table: An implementation of a set or a map using hashing is called a hash table.

Summary:

1. Chaining is a technique to implement a hash table
2. Memory consumption is O(n+m)
3. Operations work in time O(c+1)
4. Chain 的平均长度是 O（1+alpha）= O(1+n/m)
5. Want small m and small c

m 是现在 table 的 size，有的被用了，有的还是空的，比如一个 list 1000， 然后 n 是已经用了 100 个（这 100 百的后面都有 linked list，最长的 list 是 c）

Hash Functions

Load factor: Alpha = n /m ; m is cardinality of the hash function , measures how filled up is our hash table.

Good hash functions:

1. Deterministic
2. Fast to compute
3. Distributes keys well into different cells (分派平均)
4. Few Collisions

No universal hash function!!

Lemma: if number of possible keys is big(|U|>>m) for any hash function h there is a bad input resulting in many collisions. 如果 chain key 的数量大小，太多的 collision

Universal Family

Definition

Let U be the universe — the set of all possible keys. A set of hash functions

$$\mathcal{H} = \{h : U \to \{0, 1, 2, \ldots, m-1\}\}$$

is called a universal family if for any two keys $x, y \in U, x \neq y$ the probability of collision

$$Pr[h(x) = h(y)] \leq \frac{1}{m}$$

Lemma

If h is chosen randomly from a universal family, the average length of the longest chain c is $O(1+\alpha)$, where $\alpha = \frac{n}{m}$ is the load factor of the hash table.

Corollary

If h is from universal family, operations with hash table run on average in time $O(1+\alpha)$.

Ideally, load factor is between 0.5 and 1

Use O(m) = O(n/alpha) = O(n) memory to store n keys

Operations run in time O(1+alpha = O(1) on average

What if you do not the size of hash table, copy the idea of dynamic array! Resize the hash table when alpha becomes too large, Then choose new hash function and rehash all objects!.

Keep load factor below 0.9

Keep load factor below 0.9:

Rehash(T)

$loadFactor \leftarrow \frac{T.\text{numberOfKeys}}{T.\text{size}}$
if $loadFactor > 0.9$:
  Create T_{new} of size $2 \times T.\text{size}$
  Choose h_{new} with cardinality $T_{new}.\text{size}$
  For each object O in T:
    Insert O in T_{new} using h_{new}
  $T \leftarrow T_{new}, h \leftarrow h_{new}$

Copy hash value 的值从老的 table 到新的 table

Hashing Integers:

Choose prime number bigger than 10**7 比如（10,000,019）, choose hash table size with m = 1000

Lemma

$\mathcal{H}_p = \{h_p^{a,b}(x) = ((ax + b) \bmod p) \bmod m\}$
for all $a, b : 1 \leq a \leq p-1, 0 \leq b \leq p-1$
is a universal family

P = 10,000,019, a 和 b 对于不同的 hash function 不一样，x is the key (the integer number we want to hash)

The size of the hash family is p*(p-1), 因为 a 有 p-1 中选择，b 有 p 中选择

Hashing Phone Numbers

Example

Select $a = 34, b = 2$, so $h = h_p^{34,2}$ and consider $x = 1\,482\,567$ corresponding to phone number 148-25-67. $p = 10\,000\,019$.

$(34 \times 1482567 + 2) \bmod 10000019 = 407185$

$407185 \bmod 1000 = 185$

$h(x) = 185$

General Case

- Define maximum length L of a phone number
- Convert phone numbers to integers from 0 to $10^L - 1$
- Choose prime number $p > 10^L$
- Choose hash table size m
- Choose random hash function from universal family $\mathcal{H}_p$ (choose random $a \in [1, p-1]$ and $b \in [0, p-1]$)

要选取一个很大的质数 p，假如质数选取小了，有 number between 1, to 10**L-1 differ exactly by p, collision，collision 的概率大于 1/m，证明：比如 p 是很大的质数，所以(ax+b)mod p 每个值都是不一样的，collision 为 0，然后（(ax+b)mod p）mod m 这样每 m 中有个 collision(1,1001 collision，但是 1 到 1000 都不 collision），所以 collision 概率小于 1/m，单反过来 p 是不够大的数，然后比如 1,10001 做(ax+b)mod p 数值一样，肯定在 mod1000 重复的数值更多，就大于了 1/m

Proof: the average of chain is O(1+alpha) = O(1 + n/m)

Proof

- Fix key k
- For any other key l, define random variable

$$X_{kl} = \begin{cases} 1, \text{if } h(k) = h(l) \\ 0, \textit{otherwise} \end{cases}$$

一个 key 的长度取决于有几个 collision,

Proof

- Number of collisions $Y_k = \sum_{l \neq k, l \in T} X_{kl}$
- Chain length $n_{h(k)} = 1 + Y_k$
- $E(Y_k) = \sum_{l \neq k, l \in T} E(X_{kl}) \leq \sum_{l \neq k, l \in T} \frac{1}{m} \leq$ $\leq \frac{n}{m} = \alpha$
- $E(n_{h(k)}) = 1 + E(Y_k) \leq 1 + \alpha$ □

Corollary

Corollary

Using universal hashing and chaining scheme in a hash table of size m, it takes expected time $\Theta(n)$ to perform n operations of insertion, deletion, and search if there are $O(m)$ insertion operations. Thus, operations with the hash table run in amortized $O(1)$ time on average.

Corollary Proof

Proof

- $O(m)$ insertions, so $n = O(m), \alpha = O(1)$
- $1 + \alpha = O(1)$
- Expected running time of each operation is $O(1)$
- Expected running time of n operations is $\Theta(n)$

Hash on string

Hash family size p 因为 x 有 p 中选择

Polynomial Hashing

Definition

Family of hash functions

$$\mathcal{P}_p = \left\{ h_p^x(S) = \sum_{i=0}^{|S|-1} S[i]x^i \bmod p \right\}$$

with a fixed prime p and all $1 \leq x \leq p-1$ is called polynomial.

PolyHash(S, p, x)

```
hash ← 0
for i from |S| − 1 down to 0:
  hash ← (hash × x + S[i]) mod p
return hash
```

Example: $|S| = 3$

1. hash $= 0$
2. hash $= S[2] \bmod p$
3. hash $= S[1] + S[2]x \bmod p$
4. hash $= S[0] + S[1]x + S[2]x^2 \bmod p$

Lemma

For any two different strings s_1 and s_2 of length at most $L+1$, if you choose h from $\mathcal{P}_p$ at random (by selecting a random $x \in [1, p-1]$), the probability of collision $Pr[h(s_1) = h(s_2)]$ is at most $\frac{L}{p}$.

Proof idea

This follows from the fact that the equation $a_0 + a_1x + a_2x^2 + \cdots + a_Lx^L = 0 \pmod p$ for prime p has at most L different solutions x.

Cardinality Fix

For use in a hash table of zie m, we need a hash function of cardinality m.

First for string apply random h from P and then hash the resulting value again using integer hashing. Denote the resulting function by hm

可以在 P 个解中选取一个

Polynomial Hashing

Corollary

If $p > mL$, for any two different strings s_1 and s_2 of length at most $L+1$ the probability of collision $Pr[h_m(s_1) = h_m(s_2)]$ is $O(\frac{1}{m})$.

Proof

$$\frac{1}{m} + \frac{L}{p} < \frac{1}{m} + \frac{L}{mL} = \frac{1}{m} + \frac{1}{m} = \frac{2}{m} = O(\frac{1}{m}) \quad \square$$

1/m 是 hash 回 string 所 collision 的概率，L/P 是 hash 原来 hash collision 的概率

Searching Patterns

Given a Test and a pattern(word, phrase, sentence), and find all occurrences of P in T

S[I,j] substring starting from position i and end at j

Naïve Algorithm complexity is O(TP) 检查每一位 i 是不是 pattern 起点，如果是查接下来的 p 是不是符合 p，不符合的话，就返回 i+1 的位置，继续

Rabin-Karp's Algorithm: complexity 与 naïve 的一样，但是会 improve

Idea: 计算 patten 和接下来 T 中的|p|(len of p) 的 substring 是不是有一样的 hashing function 值，一样的话，继续 call AreEqual function，hashing function 不一样的话，肯定就不一样

Rabin-Karp's Algorithm

- If $h(P) \neq h(S)$, then definitely $P \neq S$
- If $h(P) = h(S)$, call $\texttt{AreEqual}(P, S)$
- Use polynomial hash family $\mathcal{P}_p$ with prime p
- If $P \neq S$, the probability $Pr[h(P) = h(S)]$ is at most $\frac{|P|}{p}$ for polynomial hashing

```
RabinKarp(T, P)
p ← big prime, x ← random(1, p − 1)
result ← empty list
pHash ← PolyHash(P, p, x)
for i from 0 to |T| − |P|:
  tHash ← PolyHash(T[i..i+|P|−1], p, x)
  if pHash ≠ tHash:
    continue
  if AreEqual(T[i..i + |P| − 1], P):
    result.Append(i)
return result
```

False Alarms

"False alarm" is the event when P is compared with $T[i..i + |P| - 1]$, but $P \neq T[i..i + |P| - 1]$.

The probability of "false alarm" is at most $\frac{|P|}{p}$

On average, the total number of "false alarms" will be $(|T| - |P| + 1)\frac{|P|}{p}$, which can be made small by selecting $p \gg |T||P|$.

Total Running Time

- If P is found q times in T, then total time spent in AreEqual is $O((q + \frac{(|T|-|P|+1)|P|}{p})|P|) = O(q|P|)$ for $p \gg |T||P|$
- Total running time is $O(|T||P|) + O(q|P|) = O(|T||P|)$ as $q \leq |T|$
- Same as naive algorithm, but can be improved!

Improving running time 两个连续的 substring 的 hash function 算法很接近，可以提前计算出所有的 hash 值，

Consecutive substrings

$T =$	a	b	c	b	d
$T' =$	0	1	2	1	3

$|P| = 3$

$h(\texttt{"cbd"}) = 2 + x + 3x^2$

$h(\texttt{"bcb"}) = 1 + 2x + x^2$

$H[2] = h(\texttt{"cbd"}) = 2 + x + 3x^2$

$H[1] = h(\texttt{"bcb"}) = 1 + 2x + x^2 =$
$= 1 + x(2 + x) =$
$= 1 + x(2 + x + 3x^2) - 3x^3 =$
$= xH[2] + 1 - 3x^3$

```
PrecomputeHashes(T, |P|, p, x)
H ← array of length |T| − |P| + 1
S ← T[|T| − |P|..|T| − 1]
H[|T| − |P|] ← PolyHash(S, p, x)
y ← 1
for i from 1 to |P|:
  y ← (y × x) mod p
for i from |T| − |P| − 1 down to 0:
  H[i] ← (xH[i + 1] + T[i] − yT[i + |P|]) mod p
return H
```

$O(|P|+|P|+|T| - |P|) = O(|T| + |P|)$

PolyHash O(|P|)

First for loop O(|P|) 算 x^^p

Second for loop runs O(|T|-|P|)

```
RabinKarp(T, P)
p ← big prime, x ← random(1, p − 1)
result ← empty list
pHash ← PolyHash(P, p, x)
H ← PrecomputeHashes(T, |P|, p, x)
for i from 0 to |T| − |P|:
  if pHash ≠ H[i]:
    continue
  if AreEqual(T[i..i + |P| − 1], P):
    result.Append(i)
return result
```

Improved Running Time

- $h(P)$ is computed in $O(|P|)$
- PrecomputeHashes runs in $O(|T| + |P|)$
- Total time spent in AreEqual is $O(q|P|)$ on average where q is the number of occurrences of P in T
- Average running time $O(|T| + (q + 1)|P|)$
- Usually q is small, so this is much less than $O(|T||P|)$

Conclusion:

1. Hash tables are useful for storing Sets and Maps
2. Possible to search and modify hash tables in O(1) on average
3. Must use good families and randomization
4. Hashes are also useful while working with strings and texts

Week 5 Binary Search Tree

Binary Search Tree:

Definition: A local Search DataStructure stores a number of elements each with a key coming from an ordered set.

RangeSearch(x, y) Returns all elements with keys between x and y

NearestNeighbors(z): Returns the element with keys on either side of z.

Add another two operations; insert, remove

Hashable is hard to search

Parts of a Tree

1. Root Node
2. Left subtree smaller keys
3. Right subtree bigger keys

X' key is than the key of any descendent of its left child, and smaller than the key of any descendant of its right child

Operations:

1. Find: Input: key k, Root R
   Output: The node in the tree of R with key k (Recursion)
2. Next: Input: Node N
   Output: The node in the tree with the next largest key
3. Range Search: Input: Numbers x, y Root R
   Output: A list of nodes with key between x and y
4. Insert Input: key k and Root

Output: Adds node with key k to the tree

5. Delete Input: Node N

Output: Removes node N from the tree

Find(k, R)

```
if R.Key = k:
  return R
else if R.Key > k :
  return Find(k, R.Left)
else if R.Key < k :
  return Find(k, R.Right)
```

Next(N)

```
if N.Right ≠ null:
  return LeftDescendant(N.Right)
else:
  return RightAncestor(N)
```

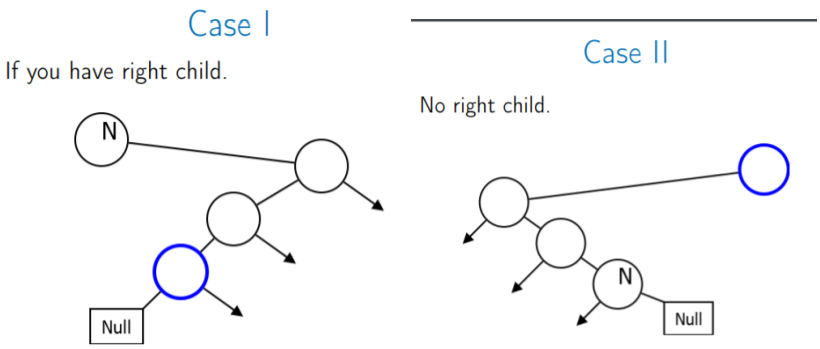
LeftDescendant(N)

```
if N.Left = null
  return N
else:
  return LeftDescendant(N.Left)
```

RightAncestor(N)

```
if N.Key < N.Parent.Key
  return N.Parent
else:
  return RightAncestor(N.Parent)
```

2．Next，最左的儿子，最右的祖先



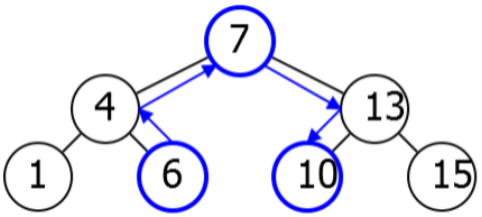
3, search;

Implementation

```
RangeSearch(x, y, R)
L ← ∅
N ← Find(x, R)
while N.Key ≤ y
  if N.Key ≥ x:
    L ← L.Append(N)
  N ← Next(N)
return L
```

Idea

RangeSearch(5, 12).



4 . Insert Input: key k and Root R

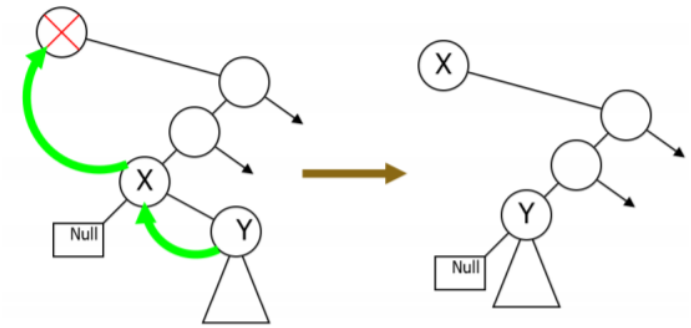
Output: Adds node with key k to the tree

```
Insert(k, R)
P ← Find(k, R)
Add new node with key k as child of
P
```

5 . Delete

假如没有 right child, 直接把 left child 作为 head

假如有 children，先寻找比 x 大一点的数，然后 把 x 的位置用 next（x）代替（next(x) 是 left descendant），next（x） 由它的 right child 代替



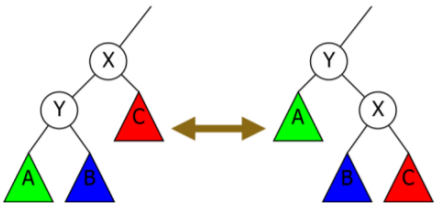

```
Delete(N)
if N.Right = null:
  Remove N, promote N.Left
else:
  X ← Next(N)
\\  X.Left = null
  Replace N by X, promote X.Right
```

For BST,

1. we want left and right subtrees to have approximately the same size.
2. Suppose Perfectly balance.
   a. Each subtree half the size of its parent
   b. Afer log2(n) levels, subtree of size 1,
   c. Operations run in O(log(n)) time

Rotation:

Rotations



$A < Y < B < X < C$

Implementation

```
RotateRight(X)
P ← X.Parent
Y ← X.Left
B ← Y.Right
Y.Parent ← P
P.AppropriateChild ← Y
X.Parent ← Y, Y.Right ← X
B.Parent ← X, X.Left ← B
```

AVL Tree:

The height of a node is the maximum depth of its subtree.

Add height field to nodes

We want the size of subtrees roughly the same. Force heights to be roughly the same

AVL Property: AVL trees maintain the following property: For all nodes N

|N.left.height – N.Right.Height|<= 1

This property implies height = O(log(n))

Theorem

Let N be a node of a binary tree satisfying the AVL property. Let $h = N.\text{Height}$. Then the subtree of N has size at least the Fibonacci Number F_h.

Large Subtrees

So node of height h has subtree of size at least $2^{h/2}$.
In other words, if n nodes in the tree, have height $h \leq 2\log_2(n) = O(\log(n))$.

Code

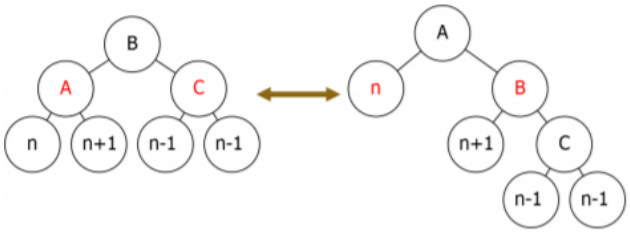
```
Rebalance(N)
P ← N.Parent
if N.Left.Height > N.Right.Height+1:
  RebalanceRight(N)
if N.Right.Height > N.Left.Height+1:
  RebalanceLeft(N)
AdjustHeight(N)
if P ≠ null:
  Rebalance(P)
```

```
AdjustHeight(N)
N.Height ← 1+ max(
                N.Left.Height,
                N.Right.Height)
```

Rebalance Left chid (root 的 left child 的比 right child 的更多) Doesn't work 当 left child 的 right child 有更多层，会将 left child 的 right child 移到 root 的 right child 的 left child，

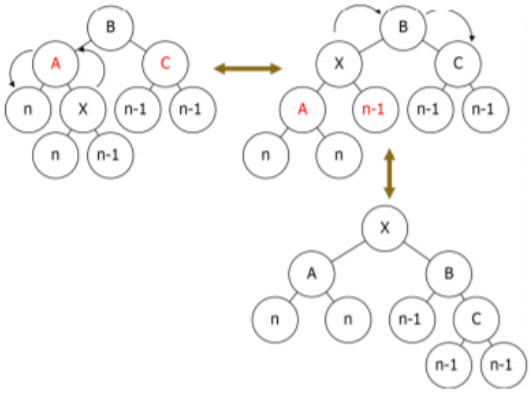
Bad Case

Doesn't work in this case.



Fix

Must rotate left first.



AVLDelete(N)

```
Delete(N)
M ← Parent of node replacing N
Rebalance(M)
```

Rebalance

RebalanceRight(N)

```
M ← N.Left
if M.Right.Height > M.Left.Height:
  RotateLeft(M)
RotateRight(N)
AdjustHeight on affected nodes
```

MergeWithRoot(R_1, R_2, T)

```
T.Left ← R1
T.Right ← R2
R1.Parent ← T
R2.Parent ← T
return T
```

Time $O(1)$.

MergeWithRoot 必须保证 R1 全部小于 R2

1. Merge: combines two binary search trees into a single one

2. Split: Breaks one binary search tree into tow

Merge

In general, to merge two sorted lists takes $O(n)$ time. However, when they are separated it is faster.

Merge

Input: Roots R_1 and R_2 of trees with all keys in R_1's tree smaller than those in R_2's

Output: The root of a new tree with all the elements of both trees

Merge

Merge(R_1, R_2)

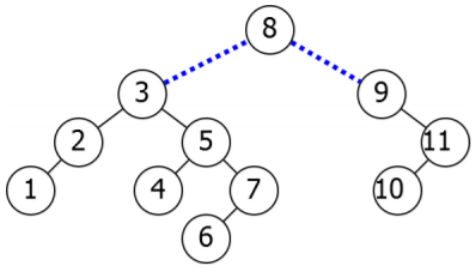
```
T ← Find(∞, R1)
Delete(T)
MergeWithRoot(R1, R2, T)
return T
```

Time $O(h)$.

Find the largest number in R1 as T

Extra Root

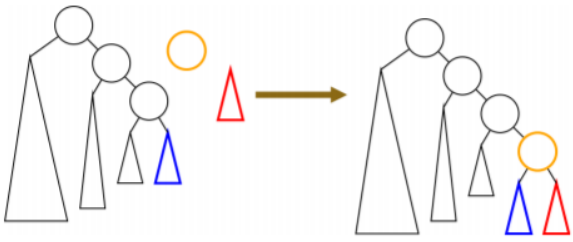
Easy if you have an extra node to add as root.



Merge 不会 preserve subtree

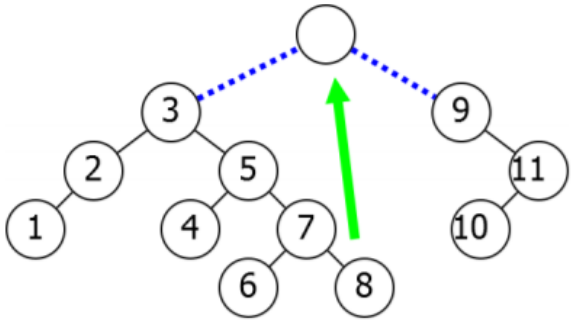
Idea

Go down side of tree until merge with subtree of same height.



Get Root

Get new root by removing largest element of left subtree.



Each step changes height difference by 1 or 2

Time O(|R1.Height - R2.Height| + 1)

AVLTreeMergeWithRoot(R_1, R_2, T)

```
if |R1.Height − R2.Height| ≤ 1:
  MergeWithRoot(R1, R2, T)
  T.Ht ← max(R1.Height, R2.Height) + 1
  return T
```

Implementation (continued)

AVLTreeMergeWithRoot(R_1, R_2, T)

```
else if R1.Height > R2.Height:
  R' ← AVLTreeMWR(R1.Right, R2, T)
  R1.Right ← R'
  R'.Parent ← R1
  Rebalance(R1)
  return root
else if R1.Height < R2.Height:
  ...
```

Implementation

Split(R, x)

```
if R = null:
  return (null, null)
if x ≤ R.Key:
  (R1, R2) ← Split(R.Left, x)
  R3 ← MergeWithRoot(R2, R.Right, R)
  return (R1, R3)
if x > R.Key:
  ...
```

Time = $\sum O(h_i - h_{i+1}) = O(h_{max}) = O(\log(n))$

Week 5 Binary Search Tree ||

N.Size returns the number of elements in the subtree of N. Should satify:

N.Size = N.left.Size + N.Right.Size + 1 Where null nodes have size zero

RecomputeSize(N)

```
N.Size ← N.Left.Size+N.Right.Size+1
```

Rotate

```
As before
RecomputeSize(Old root)
RecomputeSize(New root)
```

OrderStatistic(R, k)

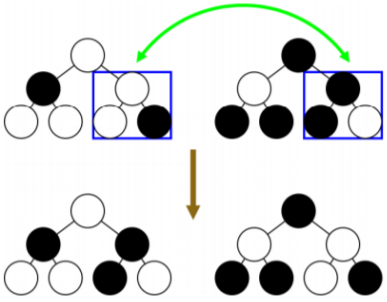
```
s ← R.Left.Size
if k = s + 1:
  return R
else if k < s + 1:
  return OrderStatistic(R.Left, k)
else if k > s + 1:
  return OrderStatistic(R.Right, k −
s − 1)
```

Flip tree colors:

One the normal color , one for opposite order

Idea

Flip using merge and split



NewArray(n)

```
Create two trees T1, T2 with keys
1...n.
Give nodes extra Color field.
All in T1 have color White
All in T2 have color Black
```

Color(m)

$N \leftarrow \texttt{Find}(m, T_1)$
return N.Color

Flip(x)

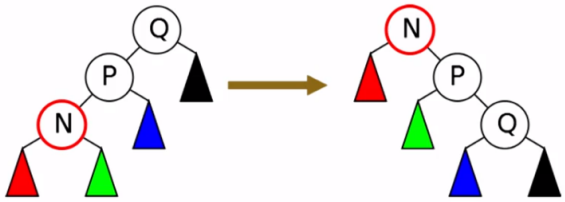
$(L_1, R_1) \leftarrow \texttt{Split}(T_1, x)$
$(L_2, R_2) \leftarrow \texttt{Split}(T_2, x)$
$\texttt{Merge}(L_1, R_2) \rightarrow T_1$
$\texttt{Merge}(L_2, R_1) \rightarrow T_2$

Splay Tree:

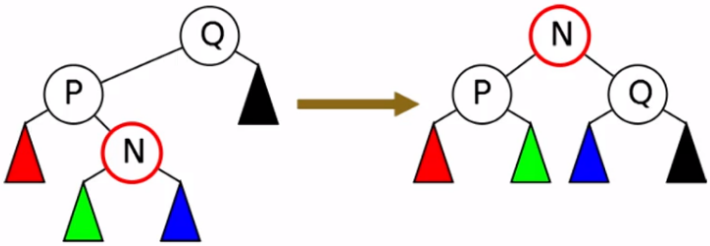
Non Uniform Inputs:

1. Search for random elements O(log(n)) best possible
2. If some items more frequent than others, can do better putting frequent queries near root.(比如 balance 的 tree root 5 在最低层，但是 unbalanced tree root 5 在第一层，所以 unbalanced 寻找更快

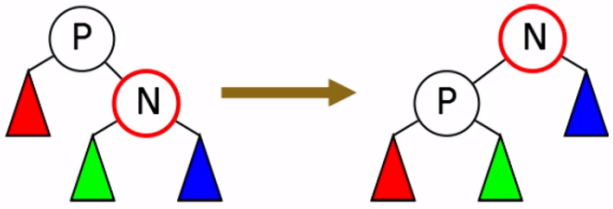
Zig-Zig



Zig-Zag



If just below root:
Zig



Splay(N)

```
Determine proper case
Apply Zig-Zig, Zig-Zag, or Zig as
appropriate
if N.Parent ≠ null:
  Splay(N)
```

STFind(k, R)

```
N ← Find(k, R)
Splay(N)
return N
```

Suppose node at depth D.

- $O(D)$ time to find N.
- Splay N.
- Amortized cost $O(\log(n))$.

You pay for the work of finding N by splaying to rebalance the tree.

Amortized Analysis

Need to amortize.

Theorem

The amortized cost of doing $O(D)$ work and then splaying a node of depth D is $O(\log(n))$ where n is the total number of nodes.

We will prove this later, but using it we can implement all our operations.

STDelete(N)

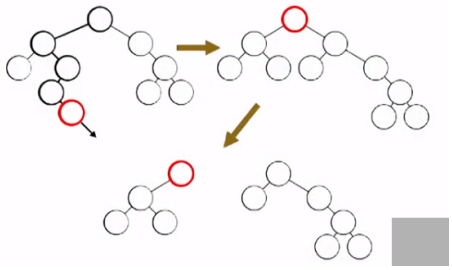
```
Splay(Next(N))
Splay(N)
L ← N.Left
R ← N.Right
R.Left ← L
L.Parent ← R
Root ← R
R.Parent ← null
```

STSplit(R, x)

```
N ← Find(x, R)
Splay(N)
if N.Key > x:
  return CutLeft(R)
else if N.Key < x:
  return CutRight(R)
else
  return N.Left, N.Right
```

Split

Splay and Cut.

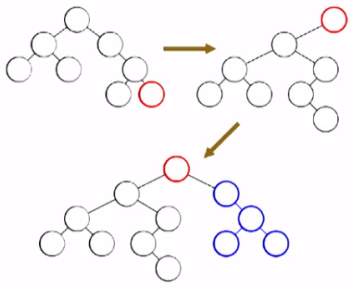


CutLeft(N)

```
L ← N.Left
N.Left ← null
L.Parent ← null
return L, N.
```

Merge

Splay largest element of first tree, and stick second tree as child of the root.



STMerge(R_1, R_2)

$N \leftarrow \texttt{Find}(\infty, R_1)$
$\texttt{Splay}(N)$
$N.\texttt{Right} \leftarrow R_2$
$R_2.\texttt{Parent} \leftarrow N$

note: after splay root do not have right tree