Algorithms

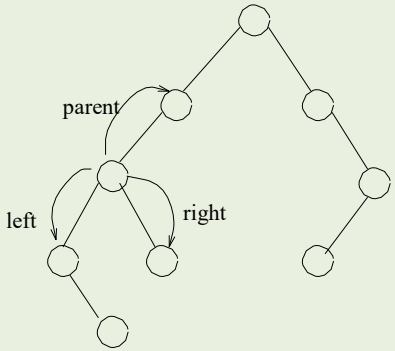
授課老師：張景堯

Binary Search Trees(二元搜尋樹)

Chapter.3-Data Structures

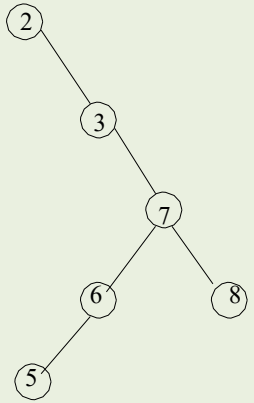
Binary Search Trees

- Binary search trees provide a data structure which efficiently supports all dictionary operations.
- A binary tree is a rooted tree where each node contains at most two children.
- Each child can be identified as either a *left* or *right* child.



Binary Search Trees

- A binary *search* tree labels each node x in a binary tree such that all nodes in the left subtree of x have keys $< x$ and all nodes in the right subtree of x have keys $> x$.



- The search tree labeling enables us to find where any key is.

Implementing Binary Search Trees

```
class BinaryTreeNode:
    def __init__(self, data):
        self.data = data
        self.parent = None #Optional
        self.left = None
        self.right = None
```

- The parent link is optional, since we can usually store the pointer on a stack when we encounter it.

Searching in a Binary Tree: Implementation

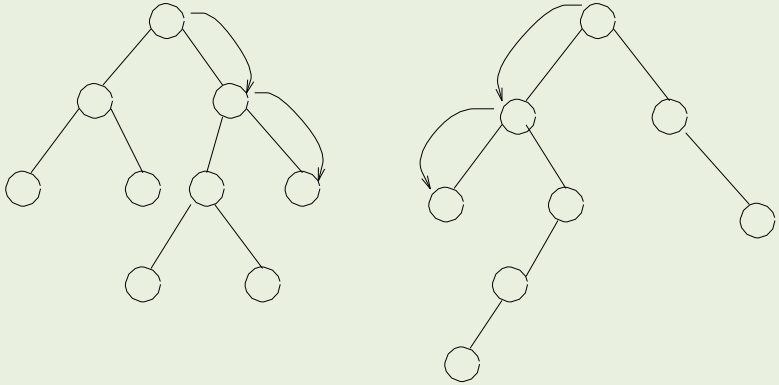
```
def search_tree(self, key):
    if self.data == key:
        return self
    if key < self.data:
        if self.left:
            return self.left.search_tree(key)
    else:
        if self.right:
            return self.right.search_tree(key)
    return None
```

Searching in a Binary Tree: How Much Time?

- The algorithm works because both the left and right subtrees of a binary search tree *are* binary search trees – recursive structure, recursive algorithm.
- This takes time proportional to the height of the tree, $O(h)$.

Maximum and Minimum

- Where are the maximum and minimum elements in a binary search tree?

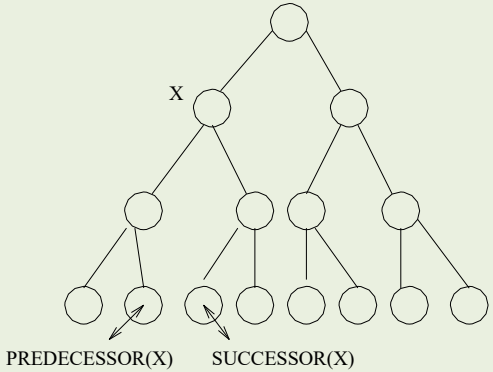


Finding the Minimum

```
def find_minimum(self):
    min_node = self
    while min_node.left:
        min_node = min_node.left
    return min_node
```

- Finding the max or min takes time proportional to the height of the tree, $O(h)$.

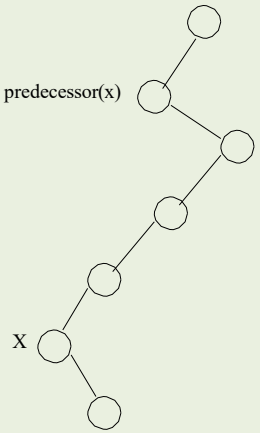
Where is the Predecessor(前導子)? : Internal Node



■If X has two children,

- its predecessor is the maximum value in its left subtree and
- its successor(後繼子) the minimum value in its right subtree.

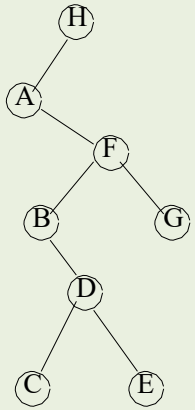
Where is the Predecessor? : Leaf Node



- If it does not have a left child, a node's predecessor is its first left ancestor. (第一個左側上代)
- The proof of correctness comes from looking at the in-order traversal of the tree.

In-Order Traversal(中序遍歷)

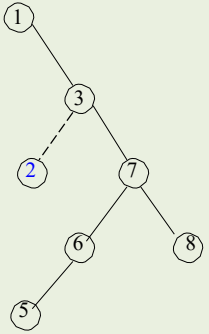
```
def tree_traversal(self):
    if self.left:
        self.left.tree_traversal()
    print(self.data, end=' ')
    if self.right:
        self.right.tree_traversal()
```



- This traversal visits the nodes in ABCDEFGH order. Because it spends $O(1)$ time at each of n nodes in the tree, the total time is $O(n)$.

Tree Insertion

- Do a binary search to find where it should be, then replace the termination NIL(None) pointer with the new item.



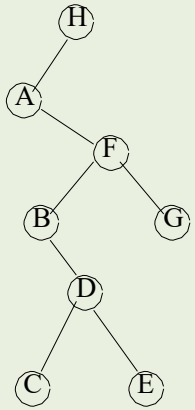
- Insertion takes time proportional to tree height, $O(h)$.

Tree Insertion Code

```
def insert_tree(self, new_node):
    if self.data > new_node.data:
        if self.left is None:
            self.left = new_node
            new_node.parent = self
        else:
            self.left.insert_tree(new_node)
    else:
        if self.right is None:
            self.right = new_node
            new_node.parent = self
        else:
            self.right.insert_tree(new_node)
```

Tree Traversal Example

```
chars = 'HAFBDGCE'
root = None
for c in chars:
    if root is None:
        root = BinaryTreeNode(c)
    else:
        root.insert_tree(BinaryTreeNode(c))

root.tree_traversal()
print()
```



[Option] Draw a Tree by Turtle-1

■使用龜圖Turtle繪製Tree

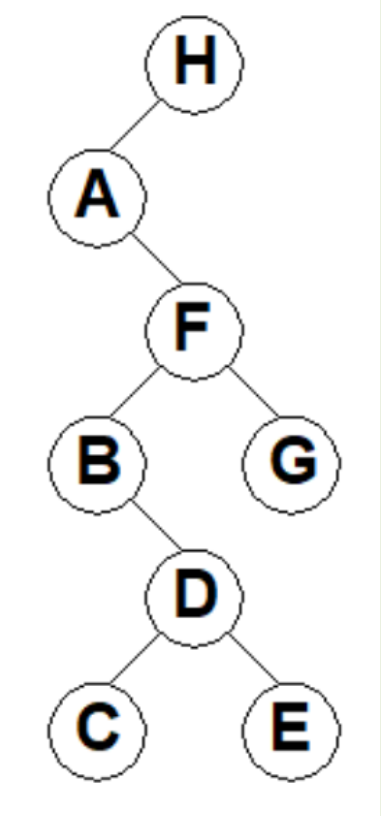
```
from turtle import *
shape('turtle')
```

```
def draw_node(x,y,text):
    up()
    goto(x-8,y-15)
    down()
    write(text, font=("Arial", 20, "bold"))
    up()
    goto(x,y-20)
    down()
    seth(0)
    circle(20)
    up()
    goto(x,y)
    down()
```

```
def draw_edge(x, y, left):
    up()
    goto(x,y-20)
    seth(0)
    if left:
        circle(20,-45)
        seth(225)
    else:
        circle(20,45)
        seth(-45)
    down()
    fd(30)
    seth(0)
```

[Option] Draw a Tree by Turtle-2

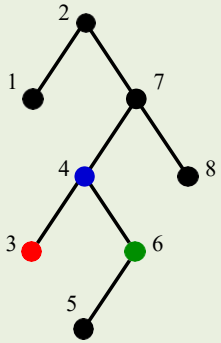
```
def draw_tree(self, x, y):
    draw_node(x, y, self.data)
    if self.left:
        draw_edge(x, y, True)
        self.left.draw_tree(x-40,y-55)
    if self.right:
        draw_edge(x, y, False)
        self.right.draw_tree(x+40,y-55)


x,y = 0,300
root.draw_tree(x, y)
hideturtle()
exitonclick()
done()
```



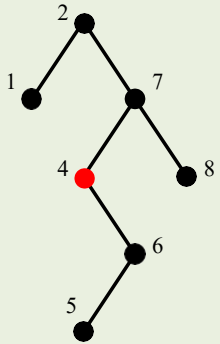
Tree Deletion

- Deletion is trickier than insertion, because the node to die may not be a leaf, and thus effect other nodes.
- There are three cases:
  - Case (a), where the node is a leaf, is simple - just NIL out the parents child pointer.
  - Case (b), where a node has one child, the doomed node can just be cut out.
  - Case (c), relabel the node as its successor (which has at most one child when z has two children!) and delete the successor!

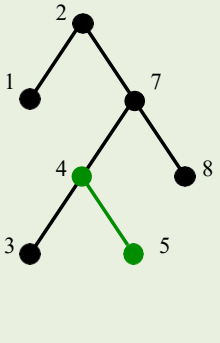
Cases of Deletion



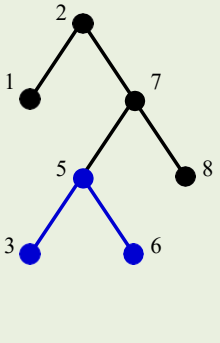
initial tree



delete node with zero children (3)



delete node with 1 child (6)



delete node with 2 children (4)
relabel as successor (5)

Binary Search Trees as Dictionaries

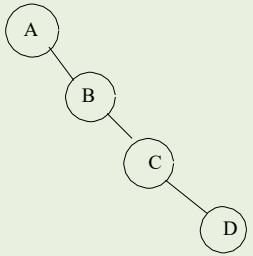
- All six of our dictionary operations, when implemented with binary search trees, take $O(h)$, where h is the height of the tree.
- The best height we could hope to get is $\log n$, if the tree was perfectly balanced, since

$$\sum_{i=0}^{\lfloor \log n \rfloor} 2^i \approx n$$

- But if we get **unlucky** with our order of insertion or deletion, we could get linear height!

Tree Insertion: Worst Case Height

insert(a)
insert(b)
insert(c)
insert(d)



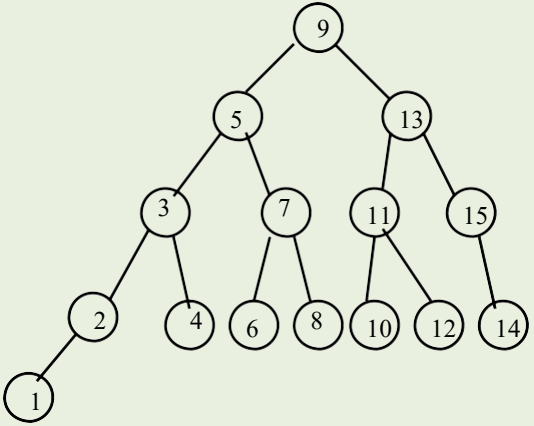
- If we are unlucky in the order of insertion, and take no steps to rebalance, the tree can have height $\Theta(n)$.

Tree Insertion: Average Case Analysis

- In fact, binary search trees constructed with random insertion orders on average have $\Theta(\log n)$ height.
- Why? Because half the time the insertion will be closer to the median key than an end key.
- Our future analysis of Quicksort will explain more precisely why the expected height is $\Theta(\log n)$.

Perfectly Balanced Trees(完美平衡樹)

- *Perfectly* balanced trees require a lot of work to maintain:



- If we insert the key 1, we must move every single node in the tree to rebalance it, taking $\Theta(n)$ time.

Balanced Search Trees(平衡搜尋樹)

- Therefore, when we talk about "balanced" trees, we mean trees whose height is $O(\log n)$, so all dictionary operations (insert, delete, search, min/max, successor/predecessor) take $O(\log n)$ time.
- No data structure can be better than $\Theta(\log n)$ in the worst case on all these operations.
- Extra care must be taken on insertion and deletion to guarantee such performance, by rearranging things when they get too lopsided.
- *Red-Black trees*, *AVL trees*, *2-3 trees*, *splay trees*, and *B-trees* are examples of balanced search trees used in practice and discussed in most data structure texts.

Where Does the Log Come From?

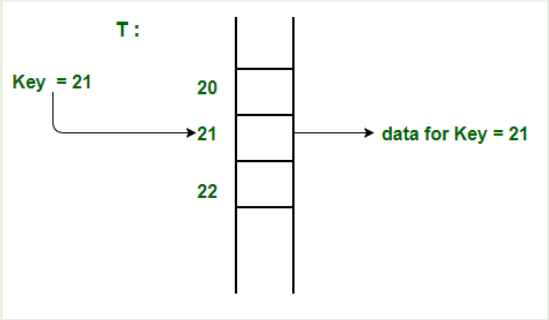
- Often the logarithmic term in an algorithm analysis comes from using a balanced search tree as a dictionary, and performing many (say, n) operations on it.
- But often it comes from the idea of a balanced binary tree, **partitioning** the items into smaller and smaller subsets, and doing little work on each of $\log(n)$ levels.
- Think binary search.

Hashing(散列 / 雜湊)

Chapter.3-Data Structures

Direct-Address Tables(直接位址表)

- Direct Address Table is a data structure that has the capability of mapping records to their corresponding keys using arrays.

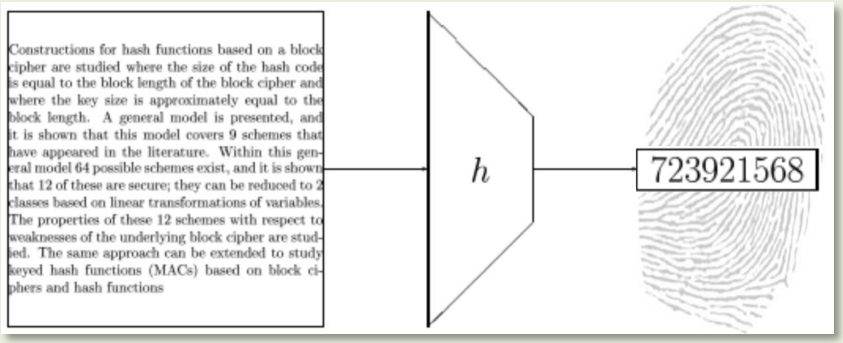


- Advantages: **Searching, Insertion, Deletion in O(1) Time,** when we have the key values (the index) that can be easily used to search/insert(put)/ delete(remove) the records in O(1) time.

- **Limitations:**
  - Prior knowledge of maximum key value
  - Practically useful only if the maximum value is very less.
  - It causes wastage of memory space if there is a significant difference between total records and maximum value.

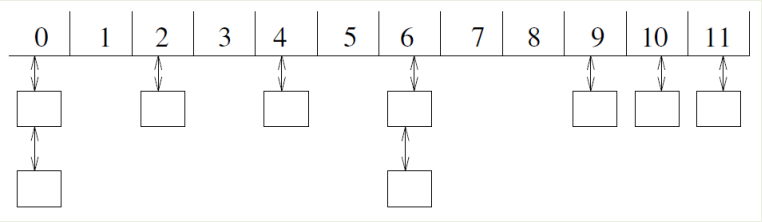
Hash Tables(散列表)

- Hashing can overcome these limitations of direct address tables.
- Hash tables are a *very practical* way to maintain a dictionary. The idea is simply that looking an item up in an array is $\Theta(1)$ once you have its index.
- A hash function is a mathematical function which maps keys to integers(hash-value).



Collisions(碰撞)

- *Collisions* are the set of keys mapped to the same bucket.
- If the keys are uniformly distributed, then each bucket should contain very few keys!
- The resulting short lists are easily searched!



- Chaining is easy, but devotes a considerable amount of memory to pointers, which could be used to make the table larger.

Hash Functions

- It is the job of the hash function to map keys to integers. A good hash function:
  1. Is cheap to evaluate
  2. Tends to use all positions from $0 \ldots M$ with uniform frequency.
- The first step is usually to map the key to a big integer, for example

$$h = \sum_{i=0}^{keylength} 128^i \times char(key[i])$$

Modular Arithmetic(模組式算術)

- This large number must be reduced to an integer whose size is between 1 and the size of our hash table.
- One way is by $h(k) = k \ mod \ M$, where M is best a large prime not too close to $2^i - 1$, which would just mask off the high bits.
- This works on the same principle as a roulette wheel!

Bad Hash Functions

- The first three digits of the Student ID Number

0	1	2	3	4	5	6	7	8	9

Good Hash Functions

- The last three digits of the Student ID Number

0	1	2	3	4	5	6	7	8	9

The Birthday Paradox 生日悖論

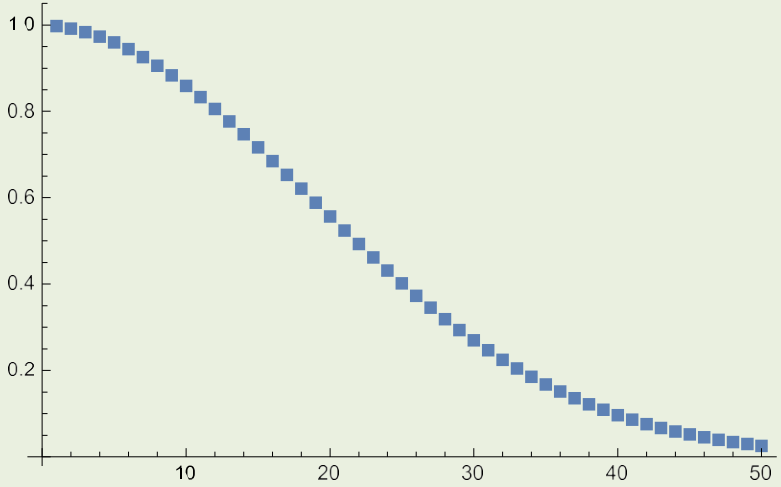
■No matter how good our hash function is, we had better be prepared for collisions, because of the birthday paradox.

■The probability of there being *no* collisions after n insertions into an m-element table is

$$(m/m) \times ((m-1)/m) \times \ldots \times ((m-n+1)/m) = \prod_{i=0}^{n-1} (m-i)/m$$

Analysis

- When $m = 366$, this probability sinks below 1/2 when $N = 23$ and to almost 0 (3%) when $N \geq 50$.



[補充] Hash Functions Design

- Perfect Hashing Function (完美散列函數)
  - Def: 此Hashing Function 絕對不會有Collision發生
  - 前提: 須先知道所有資料 (for Static Search)
- Uniform Hashing Function (均勻散列函數)
  - Def: 此種Hashing Function計算所得出的Hashing Address，對應到每個Bucket No.的機率皆相等。(不會有局部偏重的情況)
- 4 Common Hash Functions
  - Middle Square (平方值取中間位數)
  - Mod (餘數，或 Division)
  - Folding Addition (折疊相加)
  - Digits Analysis (位數值分析)

Middle Square (平方值取中間位數)

■Def: 將Key值取平方，依Hashing Table Bucket數目，選取適當的中間位數值作為Hash Address。

- e.g., 假設有1000個Bucket，範圍編號為000~999，若有一數值x = 8125，試利用Middle Square求其適當之Hash Address

Sol:

(取平方)

x = 8125 ⇨ 66015625

取中間三位 ⇨ 156 = Hash Address (取015亦可)

Mod (餘數，或 Division)

- Def: H(x) = x mod m (python: x % m)
- m的選擇之注意事項:
  - m不宜為 "2"
    - *求得的位址僅有0或1，collision的機會很大*
  - m的選擇最好是質數 (只能除盡1和除盡自己)
    - *not too close to* $2^i - 1$*, which would just mask off the high bits.*

Folding Addition (折疊相加)

- Def: 將資料鍵值切成幾個相同大小的片段，然後將這些片段相加，其總和即為Hashing Address
- 相加方式有兩種:
  - Shift (移位)
  - Boundary (邊界)
- 若有一資料x = 12320324111220，請利用兩種不同的Folding Addition方法求Hashing Address (假設片段長度為3)。

■Sol:

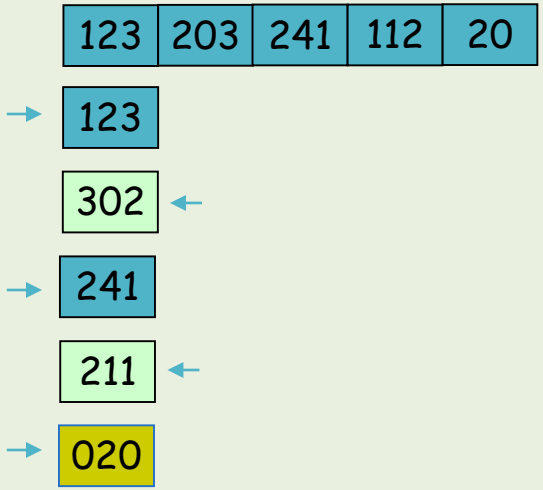
- x=12320324111220 are partitioned into three decimal digits long.

  P1 = 123, P2 = 203, P3 = 241, P4 = 112, P5 = 20.

- Shift folding:

$$h(x) = \sum_{i=1}^{5} P_i = 123 + 203 + 241 + 112 + 20 = 699$$

- Folding at the boundaries:



h(x) = 123 + 302 + 241 + 211 + 20 = 897

Digits Analysis (位數值分析)

■Def: 當資料事先已知，則可以選定基底r，然後分析每個資料之同一位數值。

- 若很集中，則捨棄該位;
- 若很分散，則挑選該位，而挑選的位數值集合成Hashing Address。

■Ex:

電話號碼	鍵值	位址
02-98[4]7[5]8[6]		456
02-98[8]7[8]6[4]		884
02-76[7]6[7]8[5]		775
02-89[2]1[6]4[3]		263
02-99[6]7[5]8[7]		657
02-88[3]7[4]8[2]		342
02-78[4]7[3]8[1]		431

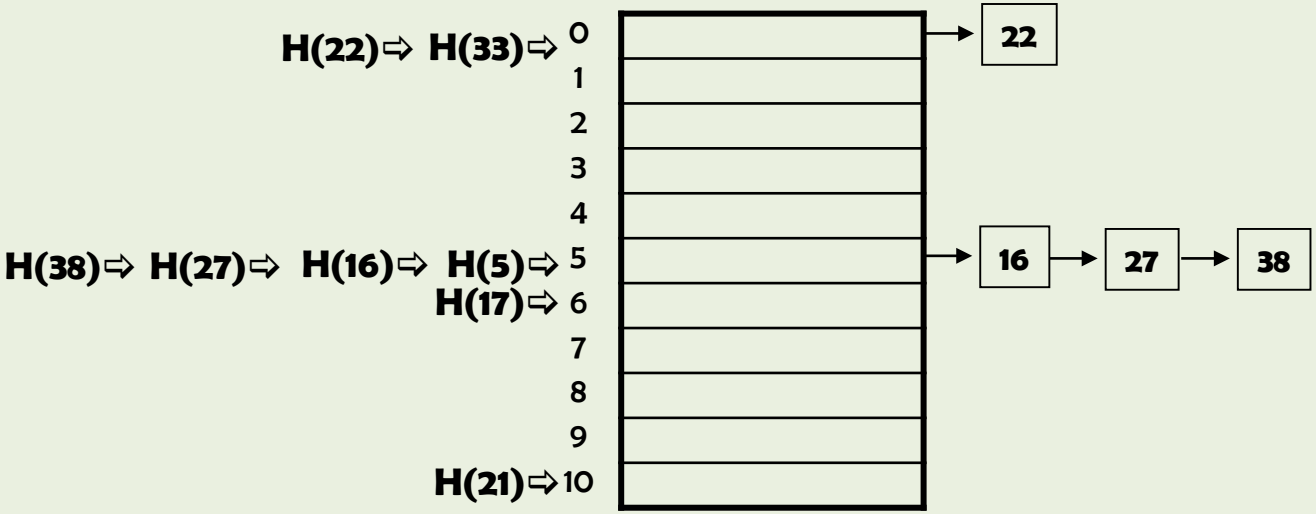
Handle Collisions

■There are mainly two methods to handle collision:

1. Separate Chaining (分離鍊接)
2. Open Addressing (開放定址)
   - Linear Probing (線性探測)
   - Quadratic Probing (二次方探測)
   - Rehashing/Double Hashing (重散列/雙倍散列)

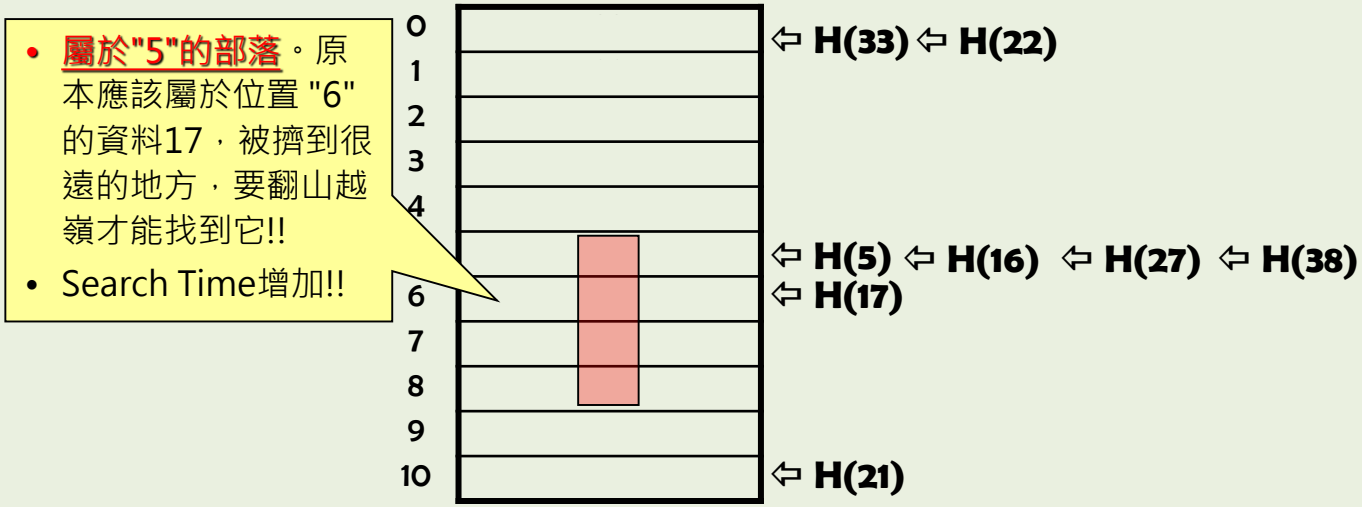
Separate Chaining (分離鍊接)

- Separate Chaining: The idea is to make each cell of hash table point to a linked list of records that have same hash function value.
  - Let us consider a simple hash function as "key mod 11" and sequence of keys as 5, 16, 33, 21, 22, 27, 38, 17.



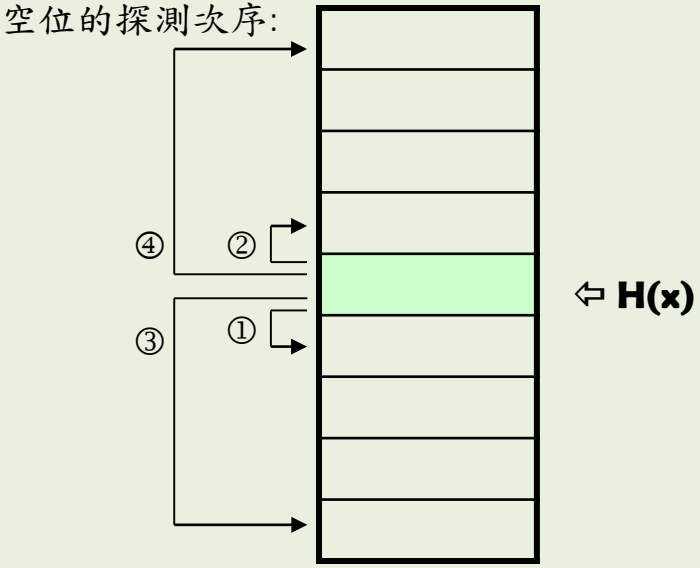
Linear Probing (線性探測)

- In linear probing, we linearly probe for next slot.
  - Let us consider a simple hash function as "key mod 11" and sequence of keys as 5, 16, 33, 21, 22, 27, 38, 17. with linear probing for collision handling.



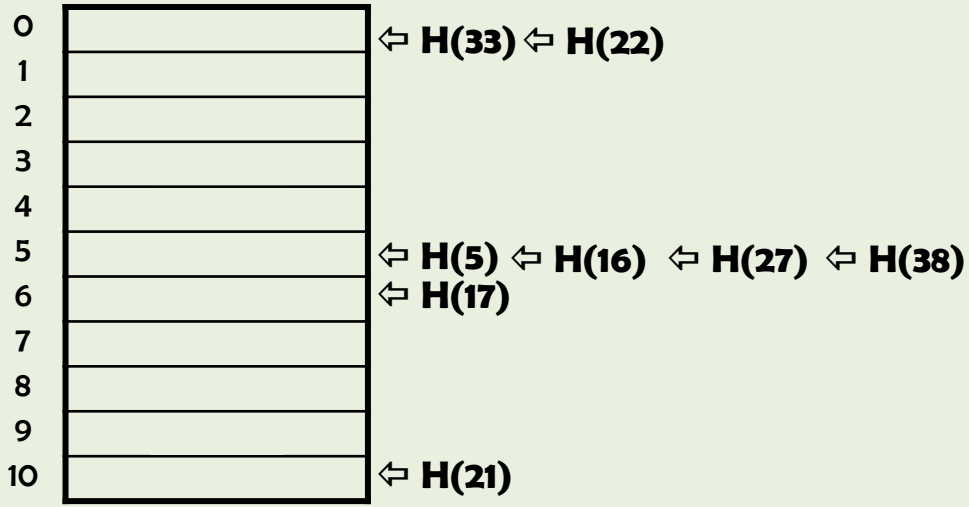
Quadratic Probing (二次方探測)

- We look for i^2-th slot in i-th iteration.
- Def: 為改善Clustering現象而提出。當H(x)發生overflow時，則探測 $(H(x) \pm i^2) \bmod b$，b為bucket數，$1 \leq i \leq (b-1)/2$



Quadratic Probing Example

- Let us consider a simple hash function as "key mod 11" and sequence of keys as 5, 16, 33, 21, 22, 27, 38, 17. with quadratic probing for collision handling.



Quadratic Probing Example

- How about 44 ⇒ ?
- Sol:

H(44) = 0 ⇨ $(0+1^2)$ mod 11 = 1

⇨ $(0-1^2)$ mod 11 = 10

⇨ $(0+2^2)$ mod 11 = 4

⇨ $(0-2^2)$ mod 11 = 7

⇨ $(0+3^2)$ mod 11 = 9

⇨ $(0-3^2)$ mod 11 = 2

負值需先加上**11的適當倍數**，再取mod!!

0	33
1	22
2	
3	
4	27
5	5
6	16
7	17
8	
9	38
10	21

Rehashing/Double Hashing (重散列/雙倍散列)

- Double hashing uses the idea of applying a second hash function to key when a collision occurs. The i-th probes double hashing can be done using :

$$(hash1(key) + i * hash2(key)) \,\%\, TABLE\_SIZE$$

- A good second Hash function is:
  - It must never evaluate to zero
  - Must make sure that all cells can be probed
- First hash function is typically

$$hash1(key) = key \,\%\, TABLE\_SIZE$$

- A popular second hash function is :

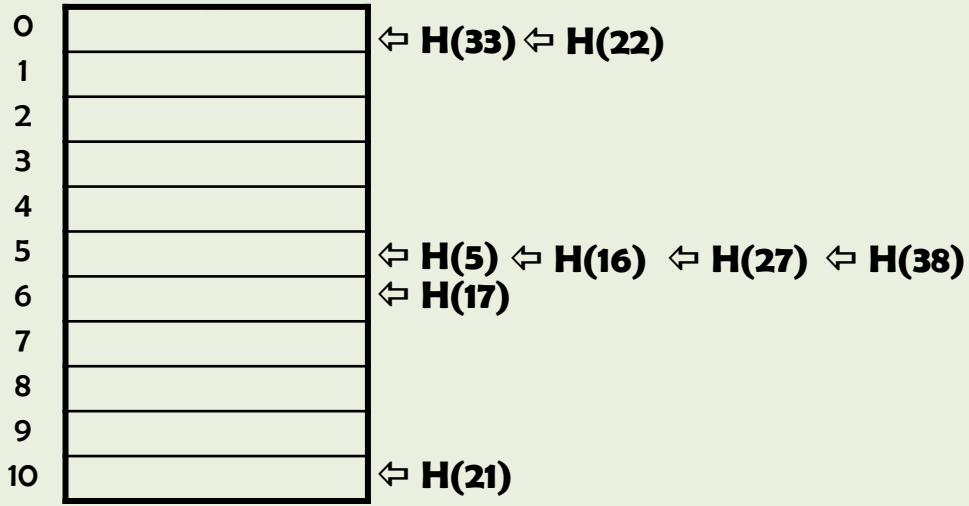
$$hash2(key) = PRIME - (key \,\%\, PRIME)$$

where $PRIME$ is a prime smaller than the $TABLE\_SIZE$.

Double Hashing Example

- Let us consider a simple hash function as "key mod 11" and sequence of keys as 5, 16, 33, 21, 22, 27, 38, 17. with double hashing using "5 - (key mod 5)" as the second hash function for collision handling.

H2(16)=4
H2(22)=3
H2(27)=3
H2(38)=2



Performance on Set Operations

- With either chaining or open addressing:
  - Search - $O(n/m)$ expected, $O(n)$ worst case
  - Insert - $O(1)$ expected, $O(1)$ worst case
  - Delete - $O(1)$ expected, $O(1)^*$ worst case

	Hash table (expected)	Hash table (worst case)
Search(L, k)	$O(n/m)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$
Delete(L, x)	$O(1)$	$O(1)$
Successor(L, x)	$O(n+m)$	$O(n+m)$
Predecessor(L, x)	$O(n+m)$	$O(n+m)$
Minimum(L)	$O(n+m)$	$O(n+m)$
Maximum(L)	$O(n+m)$	$O(n+m)$

- Min, Max and Predecessor, Successor $\Theta(n+m)$ expected and worst case. (n:data size; m:bucket size)
- Pragmatically, a hash table is often the best data structure to maintain a dictionary. However, the worst-case time is unpredictable.
- The best worst-case bounds come from balanced binary trees.

Hashing, Hashing, and Hashing

- Udi Manber says that the three most important algorithms at Google are hashing, hashing, and hashing.
- Hashing has a variety of clever applications beyond just speeding up search, by giving you a short but distinctive representation of a larger document.
  - *Is this new document different from the rest in a large corpus?* – Hash the new document, and compare it to the hash codes of corpus.(新文件是否已經有了？)
  - *Is part of this document plagiarized from part of a document in a large corpus?* – Hash overlapping windows of length w in the document and the corpus. If there is a match of hash codes, there is possibly a text match. (文件是否抄襲？)
  - *How can I convince you that a file isn't changed?* – Check if the cryptographic hash code of the file you give me today is the same as that of the original. Any changes to the file will change the hash code.(檔案還是原來的嗎？)

Substring Pattern Matching

- Input: A text string t and a pattern string p.
- Problem: Does t contain the pattern p as a substring, and if so where?
- E.g.: Is *Skiena* in the Bible?

Brute Force Search

- The simplest algorithm to search for the presence of pattern string p in text t overlays the pattern string at every position in the text, and checks whether every pattern character matches the corresponding text character.
- This runs in $O(nm)$ time, where $n = |t|$ and $m = |p|$.

String Matching via Hashing

- Suppose we compute a given hash function on both the pattern string p and the m-character substring starting from the i-th position of t.
- If these two strings are identical, clearly the resulting hash values will be the same.
- If the two strings are different, the hash values will almost certainly be different.
- These false positives should be so rare that we can easily spend the $O(m)$ time it take to explicitly check the identity of two strings whenever the hash values agree. (只需當hash值相同時再去花$O(m)$時間檢測兩字串是否相等)

The Catch 初步理解

- This reduces string matching to $n - m + 2$ hash value computations (the $n - m + 1$ windows of t, plus one hash of p), plus what should be a very small number of $O(m)$ time verification steps.
(要做$n - m + 2$即$O(n)$次hash運算，加上少數hash相同時花$O(m)$時間比對)

- The catch is that it takes $O(m)$ time to compute a hash function on an m-character string, and $O(n)$ such computations seems to leave us with an $O(mn)$ algorithm again.
(但每次都還要花$O(m)$時間做hash，這樣還是用了$O(mn)$時間，有辦法改善嗎？)

The Trick 訣竅是什麼？

- Look closely at our string hash function, applied to the m characters starting from the j-th position of string S:

$$H(S,j) = \sum_{i=0}^{m-1} \alpha^{m-(i+1)} \times char(s_{i+j})$$

- A little algebra reveals that

$$H(S,j+1) = (H(S,j) - \alpha^{m-1} char(s_j))\alpha + s_{j+m}$$

- Thus once we know the hash value from the j position, we can find the hash value from the $(j+1)$st position for the cost of two multiplications, one addition, and one subtraction. This can be done in constant time.
(除了第一次hash用$O(m)$，之後每次只要$O(1)$)

Just Hash It

- Although the worst-case bounds on anything involving hashing are dismal, with a proper hash function we can confidently expect good behavior.
- Use hashing for everything, except worst-case analysis!

Exercise

Problems of the Day

- You are given the task of reading in n numbers and then printing them out in sorted order. Suppose you have access to a balanced dictionary data structure, which supports each of the operations search, insert, delete, minimum, maximum, successor, and predecessor in $O(\log n)$ time.
  - How you can use this dictionary to sort in $O(n \log n)$ time using only the following abstract operations: insert and in-order traversal. e.g. insert * N + in-order
  - Choose the option that you can use this dictionary to sort in $O(n \log n)$ time using only the following abstract operations: minimum, successor, insert.
  - Choose the option that you can use this dictionary to sort in $O(n \log n)$ time using only the following abstract operations: minimum, insert, delete.

Problems of Day (Cont.)

- Consider the keys: 24,10,16,66,44 with hashing function "key mod 7".

1. Show the hash table that handles collisions with linear probing. (empty slot is denoted by None)
2. Show the hash table that handles collisions with $\pm$ quadratic probing. (empty slot is denoted by None)
3. Show the hash table that handles collisions with double hashing using "5-(key mod 5)" as the second hash function. (empty slot is denoted by None)

Program Exercises (Moodle CodeRunner)

- Exercise 03 (close at 3/18 23:59)
  - Two Sum: Given an list of integers, return ***indices*** of the two numbers such that they add up to a specific target.
    You may assume that each input would have exactly one solution, and you may not use the same element twice.
    - *Given nums = [2, 7, 11, 15], target = 9,*
    - *Because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1].*
  - Find the Nearest Repeated Entries: Given an list of words, return the ***distance*** between a closest pair of equal entries.
    - *Given s = ["all", "work", "and", "no", "play", "makes", "for", "no", "work", "no", "fun", "and", "no", "results"]*
    - *Because the second(s[7]) and third(s[9]) occurrences of "no" is the closest pair, 9-7=2 return 2.*