Ch05 雜湊原理

05-1 雜湊的基本原理

1.

	insertion	deletion	retrieval	traversal
binary search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
hashing	$O(1)$	$O(1)$	$O(1)$	$O(1)$

key — $\boxed{\text{Hash}}$ — location → [0] [1] [2] ⋮

2. ① Hashing : Enables access to table items in time that is relatively constant regardless of theirs locations

   ② Hash function : Maps the search key of a table item into a location that will contain the item

   ③ Hash table : An array that contains the table items, as assigned by a hash function

   ④ Each search key into a unique location
   All the search key are known] ⇒ perfect

05-2 雜湊碰撞

1. When hash function map have two or more items ⟹ defferent search key
   ⟹ the same location

2. Ex : How many people assigned 12 months such that collision is higher than 0.5?

   sol : Prob[N different months] $= \frac{11 \times \cdots \times (13-N)}{12^{N-1}}$

   Prob[23 different birthday] $= \frac{364 \times \cdots \times 343}{365^{22}} \cong 0.4927$

05-3 雜湊函數的要件

1. Assign each search key to a single location
   ① Easy and fast to compute
   ② Places items evenly throughout the hash table
   ③ Involves the entire search key
   ④ Uses a prime base, if it use modulo arithmetic

05-4 雜湊函數

1. Simple has function
   ① Digit selection ⇒ does not distribute items evenly
   ② Folding ⇒ involves the entire search key
   ③ Modulo arithmetic ⇒ the table size should be prime
   ④ Converting character strings ⇒ use integers in the hash function instead of search strings

2. Ex: ① 10027104 ⇒ 10+2+71+4 ⇒ 8+7 ⇒ 15
   ③ 10027104 ⇒ 10+2+71+4 ⇒ 87 % 13 ⇒ 9
   ④ 10027104 ⇒ 49+48+48+50+55+49+48+52 = 399 ⇒ 399 % 13 = 9
   ASCII

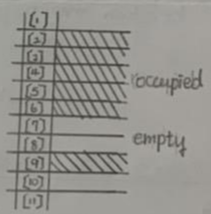
05-5 以線性探索解決碰撞

1. ① Probe for an empty location in the hash table
   ② As the hash table fills, collisions increase
   ③ Allows has table to accommodate more than one item in the same location

2. Ex: 10027146 ⇒ sum%7
   Number of Probes:
   1+1+1+2+2+4 = 11

[0] 10027112	[4] 10027224
[1] 10027236	[5]
[2] 10027246	[6] 10027102
[3] 10027146	

05-6 碰撞造成的主要群聚現象

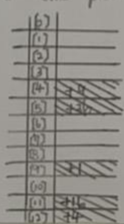


1. Linear probing ⟹ Primary clustering problem
   ① Item tend to cluster together and large cluster tend to get even larger
   ② Large culster cause long probing sequences (sequential search)
   ③ sol: Empty locations after deletions would incorrectly stop a probing sequence

05-7 以平方探索解決碰撞

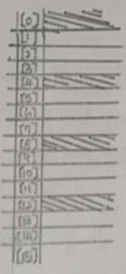
1. Search the first location and then continue at the increments of $1^2, 2^2, 3^2$ and so on
   10027207 ⇒ $5+1^2 \to [6]$ $5+3^2 \to [0]$
   $5+2^2 \to [2]$ $5+4^2 \to [0]$

2. ① May not visit every location in the hash table
   ② Different keys at the same location creat the same probing sequence
   ③ Table size must be prime



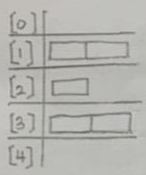
05-8 雙重碰撞

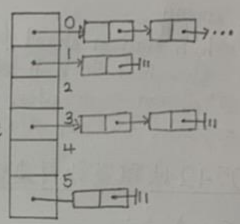
1. Step size : $h_2(key) = 5 - (sum \% 5)$
2. table size v.s. step size
   ① If they are relatively prime, the probing sequence will visit every location in the hash table
   e.g. table size = 13 , step size = 4
   table size = 16 , step size = 4



05-9 分開鏈結與多容器

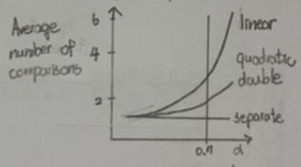
1. Buckets ⇒ Each location in the hash table is itself an array
2. Separate chaining ⇒ ① Each location is a linked list
   ② Successfully resolves collisions
   ③ The size of the hash table is dynamic
   ④ Each hash table location keeps a linked list



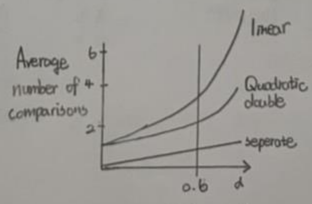


05-10 雜湊機制的效率

1. Average-case Analysis
   ① Current number of items in the table / table size
   ② Measure how full a hash table is
   ③ Unsuccessful searches generally require more time than successful searches



successful search



unsuccessful search

05-11 雜湊總結

1. Inefficient operation
   ① Traversal ⇒ Visit all the data in sorted order
   ② NN Search ⇒ Find the item that has the smallest or the largest
   ③ Range Query ⇒ Find the item between two search keys

2. Data with Multiple organization
   ① Independent ⇒ Do not support all operations efficiently
   ⇒ Waste space
   ② Interdependent ⇒ Provide a better way to support a multiple organization of data
   Enqueue V.S. Dequeue ⇒ O(n) vs. O(1)
   Hash Table ⇒ O(log n) vs. O(1)

3. Summy
   ① hash function should be extremely easy to compute and should scatter the keys evenly throughout the hash table
   ② collision occurs when two different search keys hash into the same array location
   ③ hashing does not efficiently support operations and require the items to be sorted

05-12 以雜湊執行連結查詢

1. Ex:

Accident lincese	Maintenance
MXT-612	CBC-300
LCD-123	LCD-123
KIA-214	NCC-719
NCC-719	LKK-100
MXT-500	DHT-911
LKK-100	

$6 \times 5 = 30$ comparisons

⇒

[0] → CBC-300 → LCD-123
[1] → LKK-100
[2] → NCC-719 → DHT-911
↑ probe

MXT-612[0] MXT-500[2] 5 builds
LCD-123[0] LKK-100[1] 6 probes
KIA-214[1]
NCC-719[2]

$2 \times 2 + 2 \times 1 + 2 \times 2 = 10$ comparisons

05-13 雜湊的應用

1. Ex: Notation

A 1
B 2
C 3
D 4
⋮
Y 25
Z 26

H1
[0] 2
[1] 0
[2] 3
[3] 4
[4] 7
[5] 1
[6] 2
(key*2)%7

H2
[0] 6
[1] 1
[2] 3
[3] 1
[4] 7
(key+3)%5

← CABABEFG ZXBABGZZ
3 1 2 1 2 5 6 7 26 24 2 1 2 7 26 26

← III
9 9 9

couting:
A 3
B 6
C 1
D 0
E 1
F 1
G 2

couting:
A 3
B 4
C 1
D 0
E 1
F 1
G 2

Ch06 圖形概論

06-1 七橋問題

1、V(G): vertex set (頂)
   E(G): edge set (邊)
   $G = [V, E]$
   degree: number of edges

2、vertex types ⇒ Odd or even degree (奇數 or 偶數)

06-2 七橋問題的解決

1、Eulerian path / Euler walk
   ⇒ visits every edge exactly once
   ⇒ 0 or 2 node with odd degree

2、Eulerian circuit / Eular tour
   ⇒ begin and end at the same vertex
   ⇒ 0 node with odd degree

3、$\Sigma \text{degree}(V_i) = |E(G)| \times 2$

06-3 應用於一筆畫

1、degree為偶數才能一筆畫完(一進一出)

06-4 圖形的相關術語

1、① undirected graph
   ② directed graph
   ③ adjacent vertrces (相鄰)
   ④ path
   ⑤ cycle
   ⑥ simple path
   ⑦ simple cycle

06-5 更多圖形的相關術語

1、① connected graph
   ② complete graph
   ③ strong connected graph
   ④ weighted graph

06-6 定義圖形的抽象資料型別

1. ① int numVertices;
   ② int numEdges;
   ③ int getNumVertices( );
   ④ int getNumEdges( );
   ⑤ int getWeight(Edge e);
   ⑥ void add(Edge e);
   ⑦ void remove(Edge e);
   ⑧ bool isEdge(Vertex u, Vertex v);
   ⑨ int getDegree(Vertex v);
   ⑩ bool isConnected(Graph g);
   ⑪ edgelist traverse(Graph g);
2. most common implementations of a graph
   ⇒ Adjacency matrix
   ⇒ Adjacency list

06-7 相鄰矩陣

☆ traverse(g) ⇒ $O(|V|^2)$

1. unweighted graph, matrix[i][j] is
   ⇒ 1 (true) edge exists from vertex i to vertex j
   ⇒ 0 (false) edge not exists from vertex i to vertex j
2. weighted graph, matrix[i][j] is
   ⇒ the weight of edge from vertex i to vertex j
   ⇒ ∞ (or 0) no edge from vertex i to vertex j

06-8 相鄰串列

☆ traverse(g) ⇒ $O(|V|+|E|)$

1. ① array of n linked lists
   ② i^{th} linked list has a node for vertex j if edge exists from vertex i to vertex j
2. ① Adjacency matrix ⇒ Determine whether there is an edge from vertex i to vertex j is more efficiently
   ② Adjacency list ⇒ Find all vertices adjacent to give vertex i is more efficiently
   ⇒ requires less space than adjacency matrix

06-9 循序表示法

☆ undirected graph : $|V|+2|E|+1$

1. nodes + edges

06-10 圖形走訪

1. visit all the vertices that it can reach
2. prevent indefinite loops (break the cycle)
   ⇒ mark each vertex during a visit
   ⇒ never visit a vertex more than once

06-11 深度優先走訪

1. Depth-First Search (DFS) Traversal
   ⇒ proceds along a path from a vertex as deeply into the graph as possible before backing up
   ⇒ last visited, first explored
   ⇒ has a simple recursive form
   ⇒ has an iterative form that uses a stack
2. recursiveDFS(Vetex v)

```
recursiveDFS(Vetex v)
    Mark v as visited;
    for(each unvisited vertex u adjacent to v)
        recursiveDFS(u);
```

3. interativeDFS(Vertex v)

```
interativeDFS(Vertex v)
    S.createStack( );
    S.push(v);
    Mark v as visited;
    while(! S.isEmpty( ))[
        u= S.getTop( );
        if(unvisited vertex w is adjacent to u)[
            S.push(w);
            Mark w as visited;
        ]
        else S.pop( );
    ]
```

06-12 寬度優先走訪

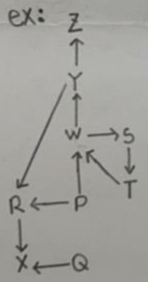
1. Breadth-First Search (BFS) Traversal
   - ⇒ visit every vertex adjacent to a vertex v before visiting any other vertex
   - ⇒ first visited, first explored
   - ⇒ iterative form uses a queue
   - ⇒ recursive form is possible, not sample

2. iterativeBFS(Vertex v)

```
q.createQueue();
q.enqueue(v);
Mark v as visited;
while(!q.isEmpty()){
  q.dequeue(u);
  for(each unvisited vertex w adjacent to u){
    Mark w as visited;
    q.enqueue(w);
  }
}
```

06-13 圖形走訪序列

1. ex:



DFS: PR RX PW WS ST WY YZ
PRXWSTYZ

BFS: PR PW RX WS WY ST YZ
PRWXSYTZ

07圖形應用

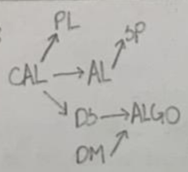
07-1初探拓樸排序

1. Topological order ⇒ directed graph without cycles
   (Acyclic Digraph or Directed Acyclic Graph, DAG)

   Topological sorting ⇒ Arranging the vertices into a topological order

07-2 拓樸排序的範例

1. Ex:



Activity-on-vertex (AOV) Network

07-3 拓樸排序的演算法版本一

1. topSort1: ① find a vertex that has no succesor (out-degree = 0)
   ② Add the vertex to the beginning of a list
   ③ Remove that vertex from the graph, as well as all edges that lead to it
   ④ Repeat the previous steps until the graph is empty

2. Ex: out-degree
   Number of Immediate successors

   Immediate predecessors

   AL → CAL
   ALGO → DS → DM
   CAL
   DM
   DS → CAL
   PL → CAL
   SP → AL

07-4 拓樸排序的演算法版本二

1. topSort 2: ① modification of iterative DFS
   ② push all vertices that no predecessor onto a stack
   ③ each time you pop a vertex from the stack, add it to beginning of a list of vertices
   ④ when end, the list of vertices will be in topological order

2. iterativeDFS(Vertex v)

```
    s.creatStack( );
    s.push(v);
    Mark v as visted;
    while(!s.isEmpty( )){
      u=s.getTop( );
      if( unvisited vertex w is adjacent to u){
        s.push(w);
        Mark w as visited;
      }
      else s.pop( );
    }
```

07-5 生成樹的簡介

1. A tree is an undirected connected graph without cycles (acyclic)
2. A spanning tree of a connected undirected graph G is
   ⇒ A subgraph of G that contains all of G's vertices and enough of its edges to form a tree
   ⇒ Application example: communication network
3. Obtain a spanning from a connected undirected graph with cycles
   ⇒ Remove edges until there are no cycles

07-6 生成樹的特性

1. Connected undirected graph ⇒ ① has n vertices must have at least n-1 edges
   ② has n vertics and exactly n-1 edges cannot contain a cycle
   ③ has n vertics and more than n-1 edges must contain at least one cycle
2. Two graphs G and H are isomorphic if and only if there is a bijection f between their vertex sets

   2 node ⇒ $2^{2-2}=1$

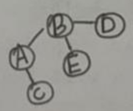
   3 node ⇒ $3^{3-2}=3$ ⇒ n^{n-2}

   4 node ⇒ $4^{4-2}=16$

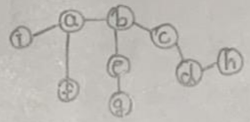
07-7 以普呂弗序列記錄一棵樹

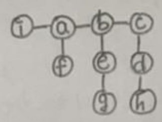
1. Each labeled tree with n vertices has a unique Prüfer sequence of length n-2
2. Conversion algorithms ⇒ Leaf with the smallest label
   ⇒ Keep the label of its parent
3. Each Prüfer sequence of length n-2 has a unique labeled tree with n vertics
4. Ex:

degree	
0	A ⟶ B ⟶ C
1	B ⟶ A ⟶ D ⟶ E
0	C ⟶ A
0	D ⟶ B
1	E ⟶ B



07-8 普呂佛序列的轉換練習

1. Ex: 



degree: [a b c d e f g h i]
2 1 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 1

07-9 以深度優先走訪建立生成樹

1. Use DFS and BFS and mark the edges that you follow
2. iterative DFS(Vertex v)

```
s.createStack( ); count=0;
s.push(v);
Mark v as visited;
while( !s.isEmpty( )&& count<|V|-1){
  u=s.getTop( );
  if( unvisited vertex w is adjacent to u){
    s.push(w); count++;
    Mark w as visited;
  }
  else s.pop( );
}
```

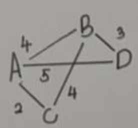
visit
T A→B→C→D
T B→A→C→D
T C→A→B
T D→A→B

DFS traversal sequence: AB BC BD
count=3

D
B
A

07-10 最小生成樹

1. Cost of spanning tree ⇒ Sum of the edge weights on a spanning tree
2. Minimum spanning tree of a connected undirected graph has a minimal edge-weight sum ⇒ A particular graph could have several minimum spanning trees



DFS: 4+4+3=11
BFS: 4+2+5=11
MST: 4+2+3=9

07-11 Prim演算法求最小生成樹

1. ① Find the least-cost edge(v, u) from a visited vertex v to some unvisited vertex u
   ② Mark u as visited
   ③ Add the vertex u and the edge(v, u) to the minimum spanning tree
   ④ Repeat the above steps until all vertices are visited

2.
```
primeAlgorithm(Vertex v)
  Mark v as visited; count=0;
  while(count<|V|-1)
    (v,u) = the least-cost edge from visited to unvisited
    Mark u as visited;
    Add(v,u) into MST;
    count++;
```

07-12 以Kruskal演算法求最小生成樹

1. ① Create a forest, where each vertex is a tree
   ② Find the least-cost edge(v, u) where vertex v and vertex u are from two different trees
   ③ Merge the trees of vertex v and vertex u, and add the edge(v,u) to the minimum spanning tree
   ④ Repeat the above steps until |V|-1 edge

2.
```
kruskalAlgorithm( )
  Assign a unique label to each vertex; count=0;
  while(count<|V|-1)
    (v,u) = the least-cost edge of two vertices with different labels
    Assign min(u,v) to all vertices with these two labels;
    Add(v,u) into MST;
    count++;
```

07-13 以sollin演算法求最小生成樹

1. Create a forest, where each vertex is a tree
2. For each tree T, do the following steps:
   ① Find least-cost edge(v, u) where vertex v is in T and vertex u is outside T
   ② Merge the trees of vertex v and vertex u, and add the edge(v, u)
3. Repeat step2 until only one tree is left
4. sollinAlgorithm( )

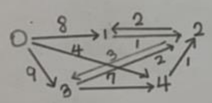
```
Assign a unique label to each vertex; size=|V|;
while(size>1)
  Initialize Edges[1...size] as empty sets;
  for each vertex v
    L=v.label;
    (v,u)=the least-cost edge from v to u for any vertex with a different label;
    if(Edges[L].weight > (v,u).weight)
      Edges[L]=(v,u);
  for each edge(v,u) in Edges but not in MST
    Assign min(v.label, u.label) to vertices in the sets of v and u;
    Add(v,u) to MST; size--;
```

07-14 初探最短路徑

1. Sortest path between two vertices in a weighted graph is the path that has the smallest sum of its edge weights

07-15 以Dijkstra演算法求最短路徑

1. Ex:



step	v	vertexSort	[0]	[1]	[2]	[3]	[4]
1	-	0	0	8	∞	9	4
2	4	0,4	0	8	5	9	4
3	2	0,4,2	0	7	5	8	4
4	1	0,4,2,1	0	7	5	8	4
5	3	0,4,2,1,3	0	7	5	8	4

07-16 最短路徑數

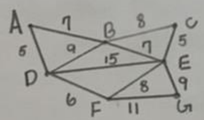
1. ① Initialize vertexSet & weight; v=v₀;
   ② Update weight each vertex u not in vertexSet, which is adjacent to V
   weight[u] = min{weight[u], weight[v] + edgeWeight[v, u]}
   ③ Find shortest path from o to u among all path stats from O, passes vertices in vertexset, and ends at a vertex not in vertexset
   if(weight[u] is minimum) vertexset = vertexset + {u};
   ④ Repeat steps 2, 3 until no more vertex can be added

2. DijkstraAlgorithm(vertex v₀)

```
weight[0..n] = {0, ∞, ... ∞};
vertexSet = φ;    v=V₀;
do{ Add v into vertexSet;
    for edge(v, u) where u is not in vertexSet
        weight[u] = min{weight[u], weight[v] + edgeWeight[v, u]};
    cheapest = ∞;
    for vertex u not in vertexSet
        if(weight[u] < cheapest){
            v=u; cheapest = weight[u];
        }
} while(cheapest < ∞)
```

07-17 Dijkstra演算法的範例

1. Ex:



vertexSet = {A, D, B}
weight = [0, 7, ∞, 5, 20, 11, ∞]

07-18 Dijkstra演算法的正確性

1. $\ell(P) \underset{\text{nonnegative weights}}{\geq} \ell(D') + \ell(x,y) \underset{\text{Inductive hypothesis}}{\geq} d(x) + \ell(x,y) \underset{\text{defn of } \pi(y)}{\geq} \pi(y) \underset{\text{Dijkstra chose v instead of y}}{\geq} \pi(v)$

07-19 Dijkstra演算法的應用

07-20 任兩點之間的最短路徑

1. Floyd-Warshall algrithm

① Initialize distance matrix D^{-1} = adjacency matrix;

② For k=0 to |V|-1

$D^{k} \leftarrow D^{k-1}$; // Add vertex k into vertexset

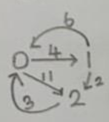
For i=0 to |V|-1

For j=0 to |V|-1

$D^{0}[1,2] = \min[D^{-1}[1,2], D^{-1}[1,0] + D^{-1}[0,2]]$

07-21 Floyd 演算法的範例

1、有向圖Ex:



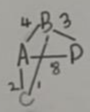
D^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

D^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

D^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

D^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

2、無向圖Ex:



D^{0}	A	B	C	D
A	0	4	2	8
B	4	0	1	3
C	2	1	0	10
D	8	3	10	0

D^{1}	A	B	C	D
A	0	4	2	7
B	4	0	1	3
C	2	1	0	4
D	7	3	4	0

D^{2}	A	B	C	D
A	0	3	2	6
B	3	0	1	3
C	2	1	0	4
D	6	3	4	0

D^{3}	A	B	C	D
A	0	3	2	6
B	3	0	1	3
C	2	1	0	4
D	6	3	4	0

D^{-1}	A	B	C	D
A	0	4	2	8
B	4	0	1	3
C	2	1	0	∞
D	8	3	∞	0

07-22.以A*演算法求最短路徑

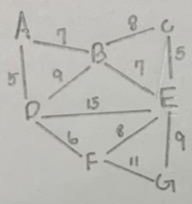
1、① BFS by keeping a priority queue and traversing a path of the lowest expected total cost

② Combines two pieces of information

⇒ Dijkstra's algorithm: favor vertices close to the origin

⇒ Greedy best-first search: favor vertices close to the goal

2. Ex：



$h(B)=14<7+9$

$h(C)=12<5+9$

$h(D)=17$

$h(E)=9$

$h(F)=11$

$f(B)=g(B)+h(B)=7+14=21$

$f(D)=g(D)+h(D)=5+17=22$

$f(D)=5+17=22$

$f(C)=g(C)+h(C)=7+8+12=27$

$f(E)=g(E)+h(E)=7+7+9=23$

$f(C)=15+12=27$

$f(E)=14+9=23$

$f(F)=g(F)+h(F)=5+6+11=22$