<시험 암기 및 정리>

Ch_5

1. If F is empty, then return
2. Traverse the subtrees of F in forest postorder
3. Traverse the remaining trees of F in forest postorder
4. Visit the root of the first tree of F

Ch_6

※ <u, v> u: tail(cause), v: head(effect)

※ complete graph:
undirected: n(n-1)/2
directed: n(n-1)

※ adjacent & incident (u, v), <u, v>
undirected:
u and v are adjacent
(u, v) is incident on u and v

directed:
u is adjacent to v
v is adjacent from u
<u, v> is incident on u on v

※ path: edge의 열거
a path from u to v
length of path is the number of edges
simple path: a path where vertices are all distinct

※ cycle:
a simple path which the first and last nodes are same

※ connected:
u and v are connected if there is a path between u and v

※ connected graph:
if all pairs of nodes in G are connected,
them G is a connected graph

※ connected component:
a maximal connected subgraph which

※ Tree: a graph which has no cycle

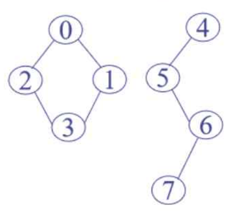
※ strongly connected:
directed graph에서 모든 node pairs에 대하여 u에서 v로 가는 path, v에서 u로 가는 path가 존재하면 그 directed graph는 strongly connected graph이다.

※ strongly connected component (clique):
a maximal subgraph that is strongly connected

※ degree:
$$e = \left(\sum_{i=0}^{n-1} d_i\right)/2$$
in-degree: edge where v is head
out-degree: edge where v is tail

※ adjacency matrix O(n^2) = $n^2 - n$
※ adjacency list

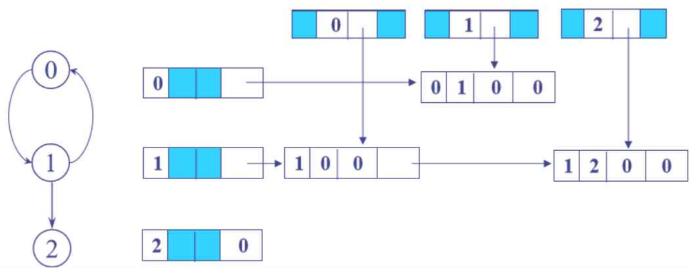
- where $0 \le i < n$



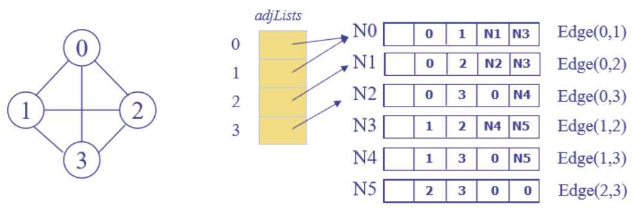
int nodes[n+2*e+1]

[0] 9	[8] 23	[16] 2
[1] 11	[9] 2	[17] 5
[2] 13	[10] 1	[18] 6
[3] 15	[11] 3	[19] 4
[4] 17	[12] 0	[20] 5
[5] 18	[13] 0	[21] 7
[6] 20	[14] 3	[22] 6
[7] 22	[15] 1	

tail	head	Column link for head	Row link for tail



※ adjacency multilist



※ Activity Networks
processor -> successor

Ch_7
※ sequential search -> 일일이 하나씩 찾음 -> O(n)
※ binary search -> order를 이용하여 찾음 -> O(log n)
※ list 비교 -> no order O(mn)
※ list 비교 -> order O(max(nlogn, mlogm))

※ decision tree
-> input length = n, n! possible outputs
-> n! leaves
-> $\log_2(n!)+1$ height

※ merge sort
recursive: 1을 기준으로 큰 덩어리를 자른다.
iteravive:

(십진수, 3자리수 기준)
※ radix sort의 TC는 O(d(n+r))

Method	Worst	Average
Insertion sort	n^2	n^2
Heap sort	$n\log n$	$n\log n$
Merge sort	$n\log n$	$n\log n$
Quick sort	n^2	$n\log n$

※ radix sort에서 r은 10, d는 3, n은 element의 수이다.

Ch_8
※ overflow: pair를 넣을 때 bucket이 이미 꽉 차 있을 때를 의미
※ collision: pair를 넣을 때 bucket에 이미 뭐가 있을 때를 의미

※ Division: h(n) = ((k)%(prime#))%b
※ Mid-Square: 제곱해서 중간을 자름 -> r자릿수이면 0~2^{r-1}가 h(k)의 범위이다.
※ Folding(shift folding): 1234567890129384->123+456+789+012+938+4
※ Folding(Folding at boundaries):
1234567890129384->123+456+789+012+938+4->123+654+789+210+938+4

※ overflow handling (Open addressing)
->
※ linear proving -> ht[(h(k)+i)%b] (0 <= i <= b-1)
-> expected key comparisons = p =(2-a)/(2-2a), a=n/sb
-> 이미 가득 차 있으면 ht를 두배로 늘림
※ quadratic proving -> ht[(h(k)+i^2)%b] (0 <= i <= (b-1)/2)
※ Rehashing: 여러 hash함수를 놓고, overflow되면 다른 걸 쓴다.

※ overflow handling (Chaining)

->the number of comparisons needed to search: /n

[0] → acos atoi atol
[1] → NULL
[2] → char ceil cos ctime
[3] → define
[4] → exp
[5] → float floor
[6] → NULL
...
[25] → NULL

1: acos, char, define, exp, float

2: atoi, ceil, floor

3: atol, cos

4: ctime

-> 1*5+2*3+3*2+4*1=5+6+6+4=21

21/n = 21/11

※ Dynamic hashing

k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C4	110 101

-> h(C4,4)=0101(2)=5

h(k, p) -> $b=2^p$, size of directory = 2^p, directory depth = p

overflow되면 bucket을 복제한다.

overflow된 bucket을 split하고, pointer를 duplicate한다.

Ch_9

single-ended priority queue:

return minimum/maximum element -> O(1)

Insert arbitrary element -> O(log n)

Delete minimum/maximum element ->O (log n)

double-ended priority queue:

return minimum element

return maximum element

Insert arbitrary element

Delete minimum element

Delete maximum element

meld operation은 O(n)시간이 걸리는데, Leftist tree를 이용하면 O(log n)이 걸린다.

※ shorest(leftChild(x))>=shorest(rightChild(x))
※ n ≥ (2^(shortest(root)))-1
※ shortest(root) ≤ $\log_2(n+1)$

※ Weight-Biased Leftist Trees
-> 아래로 내려갈수록 node의 수가 적어진다는 보장이 있다.
Height-Biased Leftist Trees
-> top에서 bottom으로, bottom에서 top으로 가는 two step이 필요하다.
-> Weight-Biased Leftist Trees는 아래로 내려갈수록 node의 수가 적어진다는 보장이 있어서 one step만 하면 된다.