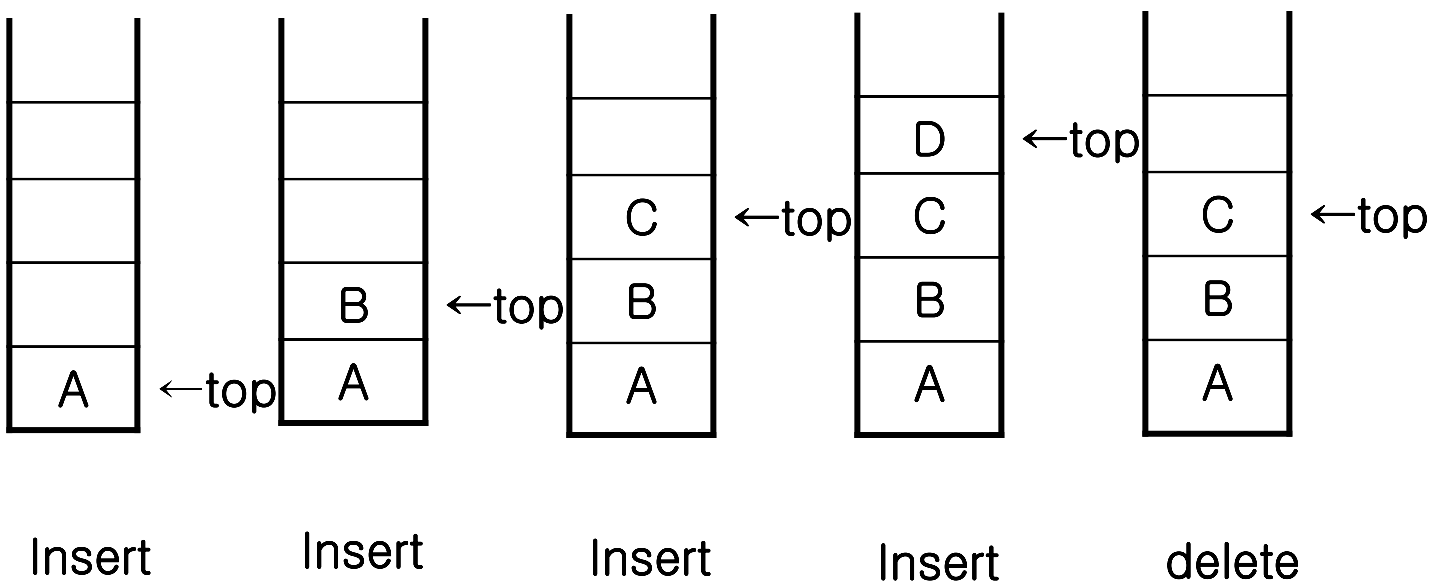
기본 자료구조

1. Stack and Queue
2. Linked List
3. Tree
4. Graph

stacks and Queues

스택의 정의

- 정의 : 톱(top)이라고 하는 한 끝에서 모든 삽입과 삭제가 일어나는 순서 리스트
- 후입 선출(last-in-first-out : LIFO) 리스트라고도 함.



스택의 추상 데이터 타입

Structure Stack

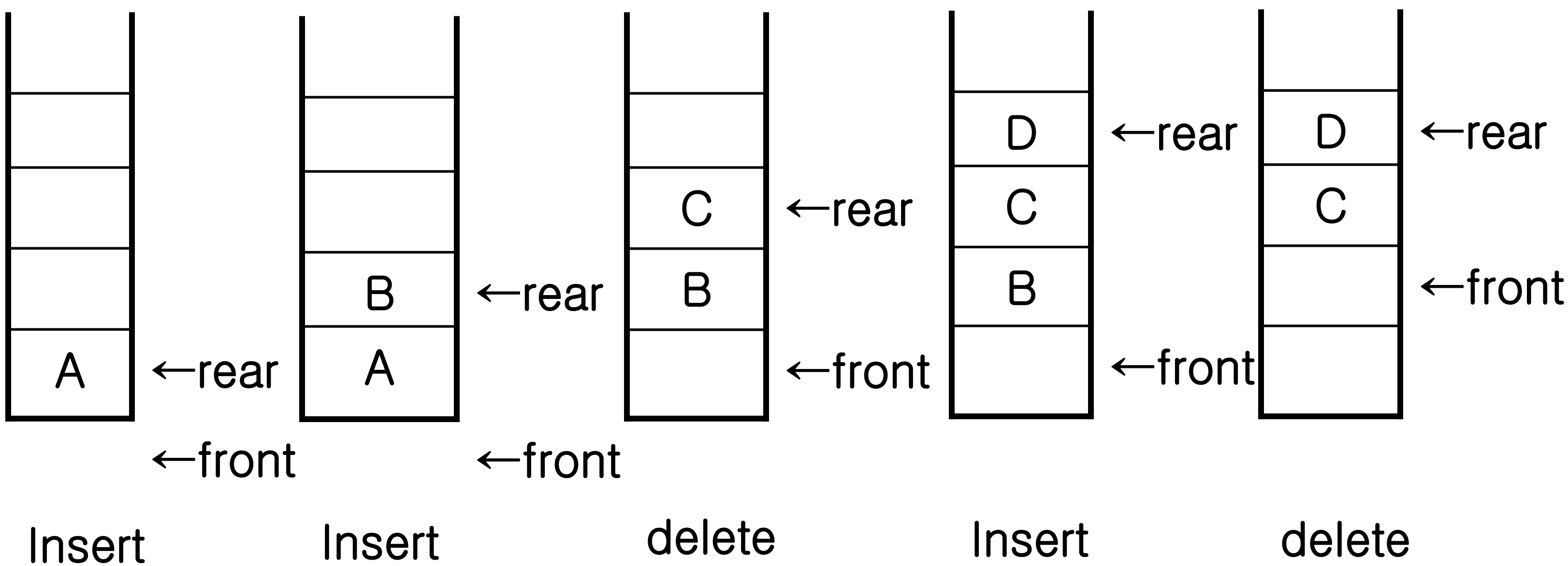
objects : 0개 이상의 원소를 가진 유한 순서 리스트

functions :

```
모든 stack ∈ Stack, item ∈ element, max_stack_size ∈ integer
    top ∈ integer
    Stack CreateS(max_stack_size)
    Boolean IsFull(stack, max_stack_size)::= if top >= max_stack_size-1
    Stack Add(stack, item) ::= 스택의 톱에 item을 삽입
    Boolean IsEmpty(stack) ::= if top < 0
    Element Delete(stack) ::= 스택에서 톱의 item을 제거
end Stack
```

큐의 정의

- 정의 : 한쪽 끝에서 데이터가 삽입되고 반대쪽 끝에서 삭제가 일어나는 순서 리스트
- 선입선출(first-in-first-out:FIFO)리스트라고 함.



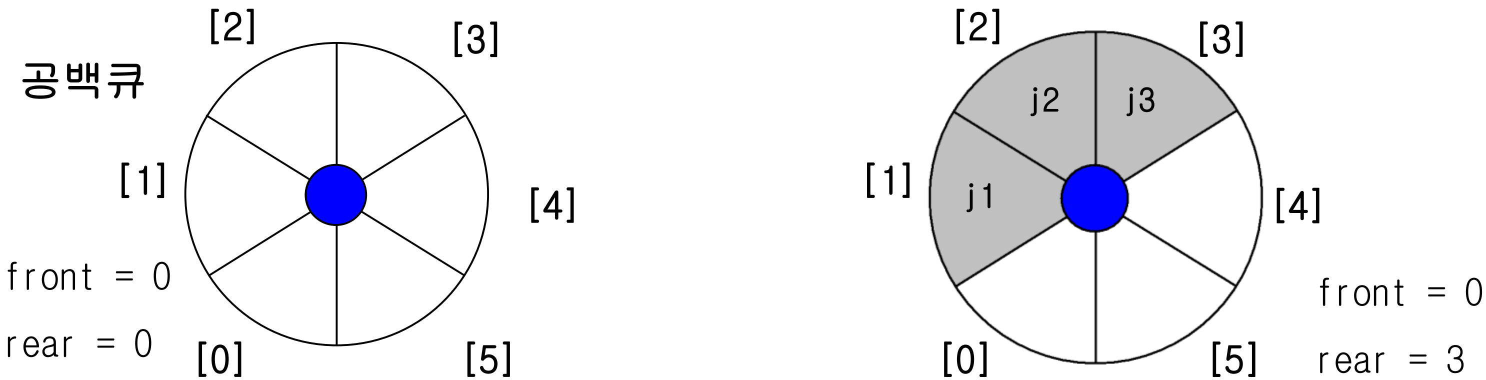
큐의 추상데이타 타입

```
Structure Queues
    objects : 0개 이상의 원소를 가진 유한 순서 리스트
    functions :
        모든 queue ∈ Queue, item ∈ element,
   max_queue_size ∈ integer front, rear ∈ interger
        Stack CreateQ(max_queue_size)
        Boolean IsFullQ(queue, max_queue_size)
        ::= if rear >= max_queue_size - 1
        Stack AddQ(queue, item)
        Boolean IsEmptyQ(queue) ::= if front == rear
        Element DeleteQ(queue)
    end Queue
```

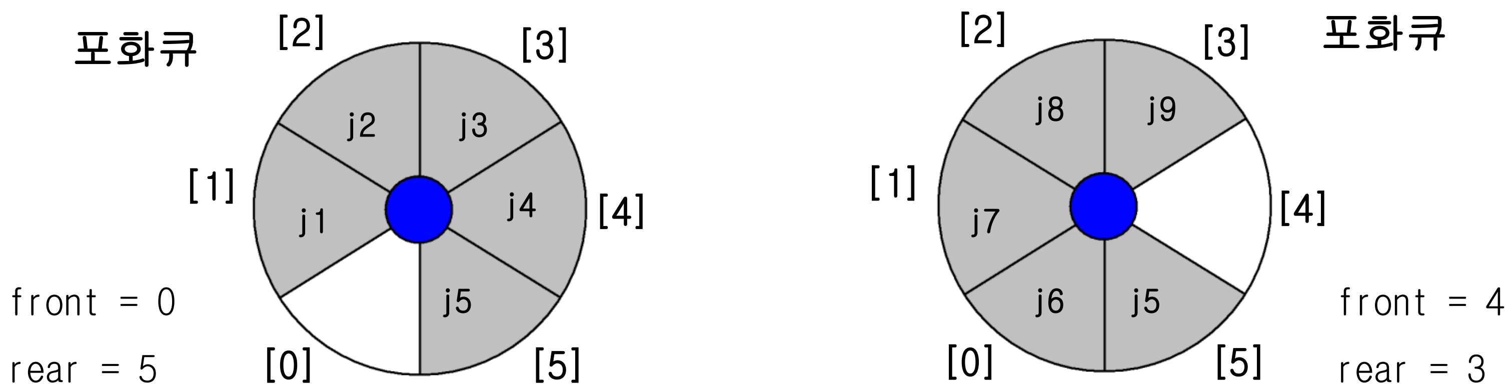
원형 큐 (Circular queue)

- 큐의 효율적인 사용
- rear = (rear + 1) % MAX_QUEUE_SIZE
- front = (front + 1) % MAX_QUEUE_SIZE
- 포화상태와 공백상태를 구별하기 위해 하나의 여유 공간(queue[rear])을 사용
- 원형 큐에서는 최대 MAX_QUEUE_SIZE –1 개의 원소만 허용

원형큐 (Circular queue) Cont'd



공백 원형 큐와 공백이 아닌 원형 큐



포화 원형 큐

Linked Lists

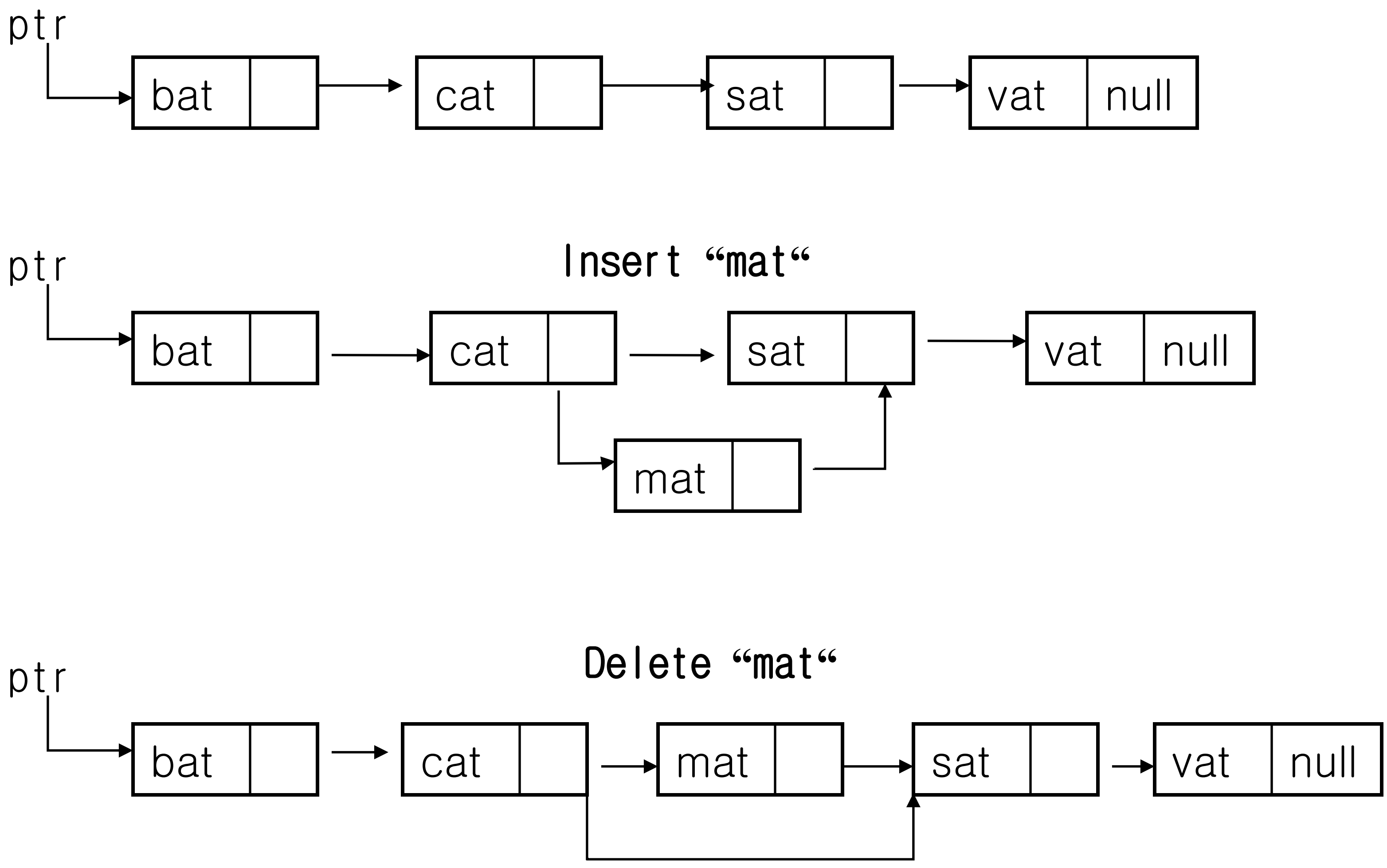
단순 연결 리스트

- 크기가 변하는 (삽입, 삭제) 순차리스트를 요구 할때 배열과 같은 순차적인 표현은 부적당하다.

- 특징
  - 노드들은 메모리의 순차적 위치에 존재하지 않는다.
  - 노드들의 위치는 실행시마다 바뀔 수 있다.

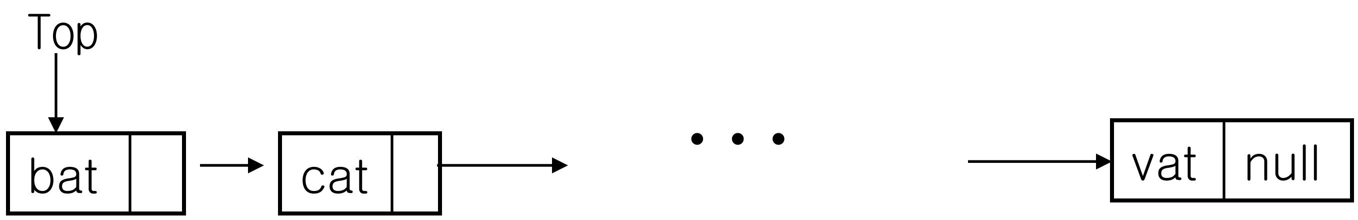
- 구조 : 연결 리스트의 노드는 데이타부(data field)와 연결부(link field)로 구성된다.
  - 각각은 여러 개의 서브부로 나누어질 수 있다.

단순 연결 리스트의 일반적인 표현

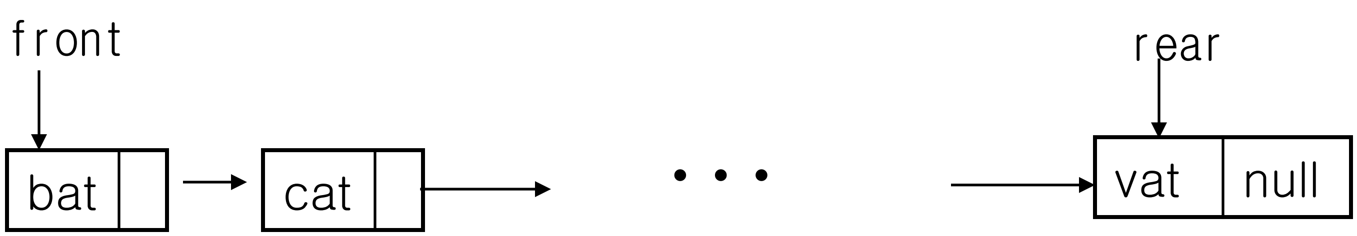


연결 스택과 큐

- 여러 개의 스택과 큐를 순차적으로 표현하여 효율적으로 관리할 수 있다.



STACK

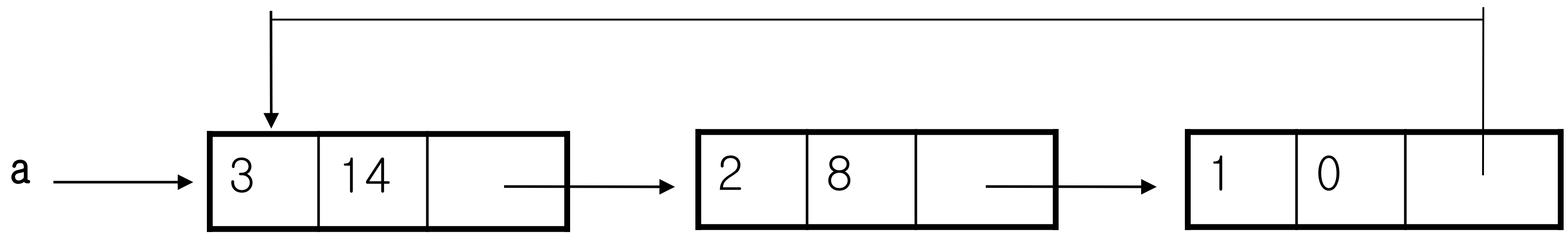


QUEUE

원형 연결 리스트

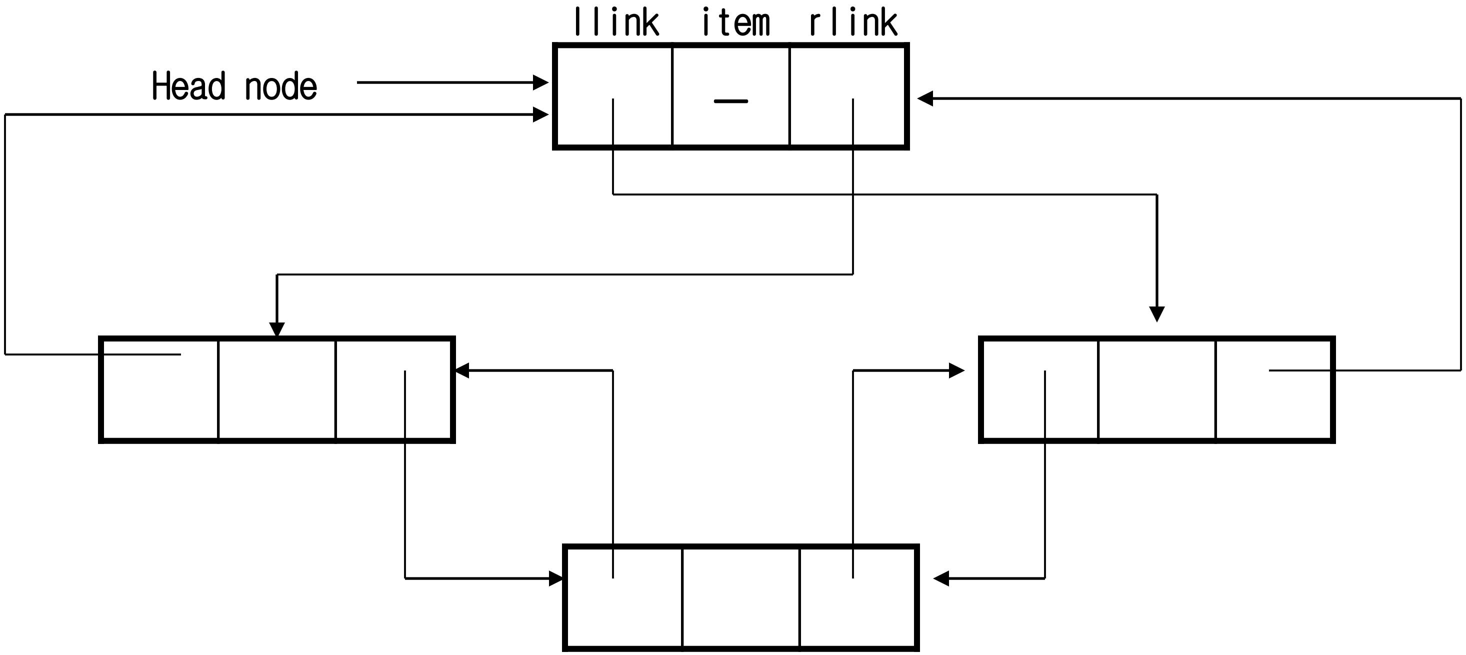
- 원형 리스트(circular list) : 마지막 노드가 리스트의 첫번째 노드를 가리키도록 한 경우

- 체인(chain) : 마지막 노드의 링크 필드 값이 NULL인 단순 연결 리스트



프로그램 3.8 3.9

이중 연결 리스트 구조



Trees

트리의 개요

- 정의 : 트리는 1개 이상의 노드로 이루어진 유한 집합으로써
  1. 노드 중에는 루트(root)라고 하는 노드가 있다.
  2. 나머지 노드들은 n≥0개의 분리집합 T1, … , Tn으로 분리될 수 있다.
     여기서 T1, … , Tn은 각각 하나의 트리이며 루트의 서브 트리라고 한다.

- 차수(degree) : 노드의 서브 트리의 수
  - A의 차수는 3, F의 차수는 0

- 리프(leaf)노드 또는 단말(terminal)노드 : 차수가 0인 노드(F, G, I, J…)

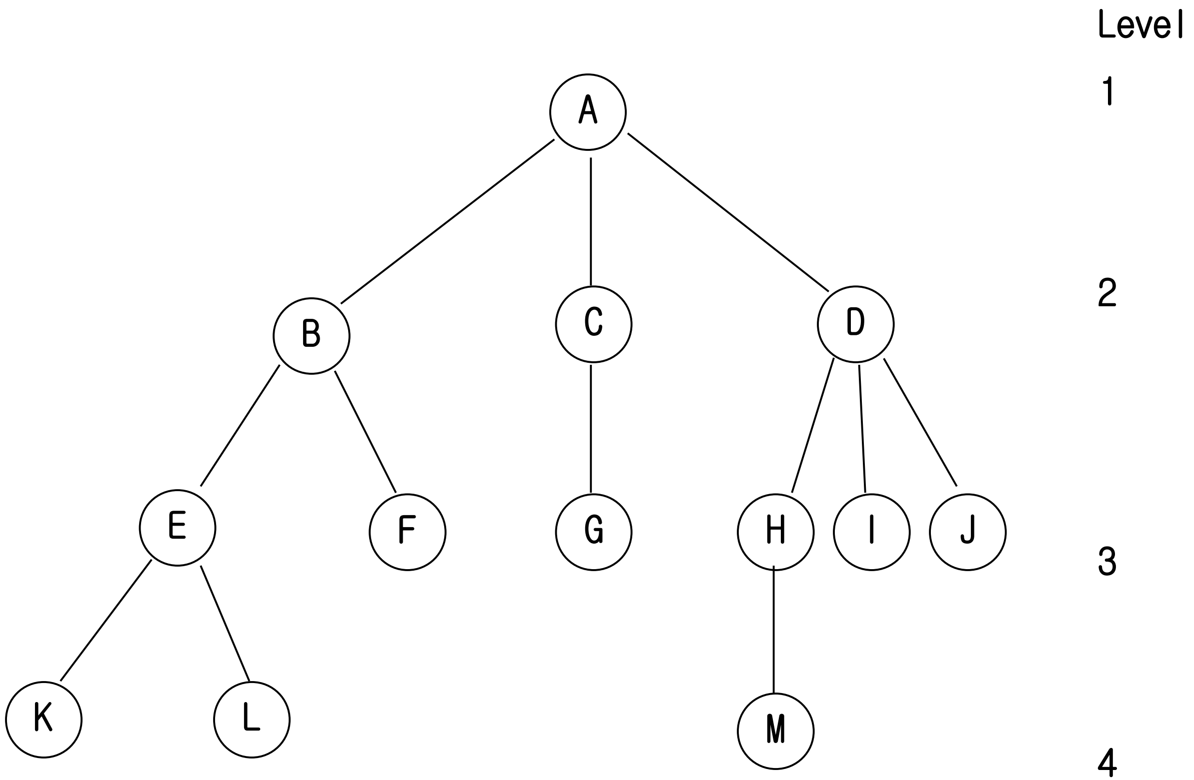
- 트리의 높이(height)또는 깊이(depth) : 트리에 속한 노드의 최대 레벨

트리의 개요

- 부모(parent)노드와 자식(child)노드

- 조상(ancestors)노드와 자손(descendants)노드
  - M은 조상은 A, D, H이고 E, F, K, L은 B의 자손

샘플 트리



트리의 표현

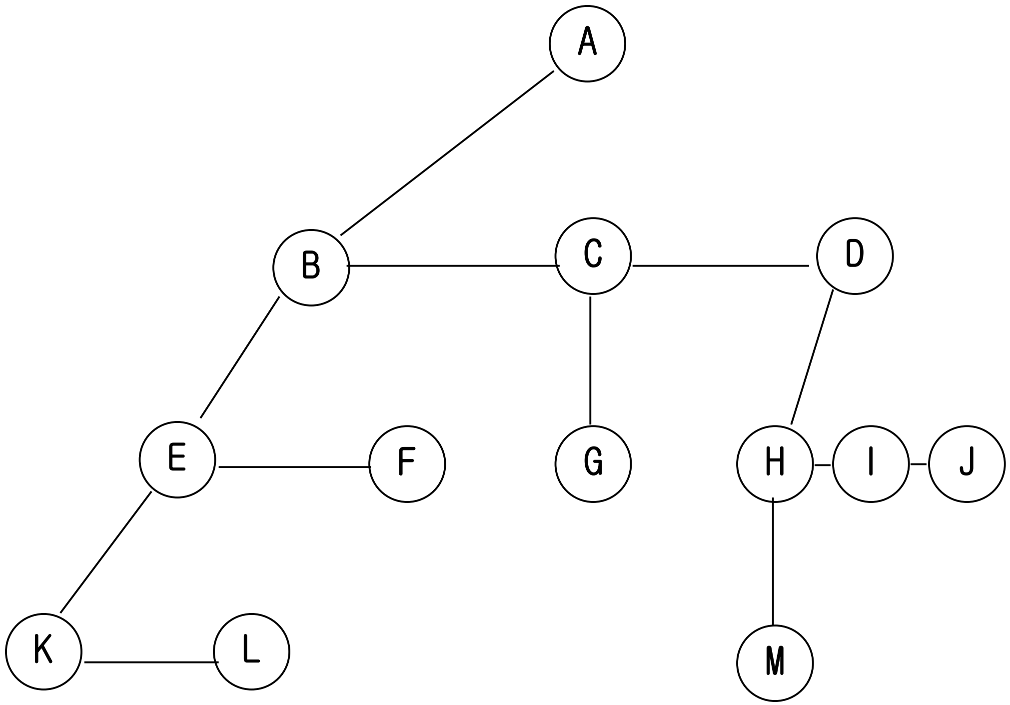
- 리스트의 표현

데이타	링크1	링크2	...	링크n

- 왼쪽 자식-오른쪽 형제 표현

데이터	
왼쪽 자식	오른쪽 형제

트리의 표현2



이진 트리

- 모든 노드의 차수가 ≤ 2인 트리
- 정의 : 공집합이거나 루트와 왼쪽 서브트리, 오른쪽 서브트리라고 부르는 두개의 분리된 트리로 구성된 노드의 유한 집합

이진 트리의 특성

- 경사 트리(skewed tree), 완전이진 트리(complete binary tree)
- 정리 1[최대 노드수]
  1. 이진 트리의 레벨 i에서 최대 노드수는 $2^{i-1}(i\geq1)$
  2. 깊이가 k인 이진 트리가 가질 수 있는 최대 노드수는 $2^k-1(k\geq1)$
- 정리 2[단말 노드수와 차수가 2인 노드수와의 상관관계] : 모든 이진 트리 T에 대하여,
  n_0는 단말 노드수, n_2는 차수가 2인 노드수라고 하면 $n_0 = n_2 + 1$이다.
- 정의[포화 이진 트리(full binary tree)] : 깊이가 k, 노드수가
  $2^k-1(k\geq0)$인 이진트리.

이진 트리의 표현

- 배열의 표현
  - 정리 3 : n개의 노드를 가진 완전 이진 트리(깊이 = $[\log_2 n+1]$)가 순차적으로 표현되어 있다면

  - (1) $i \neq 1$ 이면 parent(i)는 [i/2]의 위치에 있게 된다.
    만일 i=1 이면 i는 루트이다.
  - (2) $2i \leq n$ 이면 left_child(i)는 2i의 위치에 있게 된다.

  - (3) $2i + 1 \leq n$ 이면 right_child(i)는 2i+1의 위치에 있게 된다.

- 링크 표현

Left_child	data	Right_child

이진 탐색 트리

- 임의의 원소에 대한 삭제와 삽입, 탐색연산을 효율적으로 하기위한 트리구조

- 정의 : 공백이 가능한 이진 트리이고 공백이 아닌 경우 아래의 성질을 만족한다.

  1. 모든 원소는 키를 가지며 키는 유일한 값이다.
  2. 왼쪽 서브 트리에 있는 키들은 서브트리의 루트 키보다 작아야한다.
  3. 오른쪽 서브 트리에 있는 키들은 서브 트리의 루트 키보다 커야한다.
  4. 왼쪽과 오른쪽 서브트리도 이진 탐색트리이다.

히프 트리

- 최대 트리(max tree) : 각 노드의 키 값이 그 자식의
  키값보다 작지 않은 트리
- 최대 히프(max heap) : 최대 트리인 완전 이진 트리,
- 최소 트리(min tree) : 각 노드의 키 값이 그 자식의 키값보다
  크지 않은 트리
- 최소 히프(min heap) : 최소 트리인 완전 이진 트리,
- 정의 : 최대 히프 구조를 이용하여 가장 큰 레코드를 하나씩
  히프에서 추출한다

Graph

정의

- 정의 : 그래프 G는 공집합이 아닌 정점(vertex)들의 유한 집합(V(G))과 공집합을 포함한 간선(edge)들의 유한 집한(E(G))으로 구성

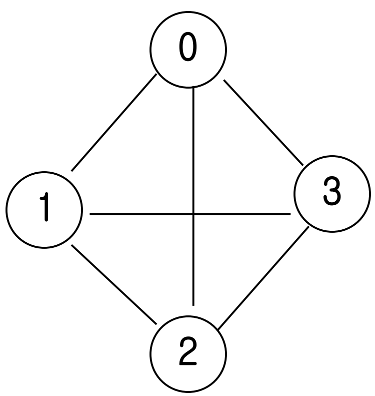
- 그래프 G=(v,E)로 표기

- 무방향 그래프(undiredted graph) : 방향이 없는 간선,(v_0,v_1)과 (v_1,v_0)은 같은 간선이다.

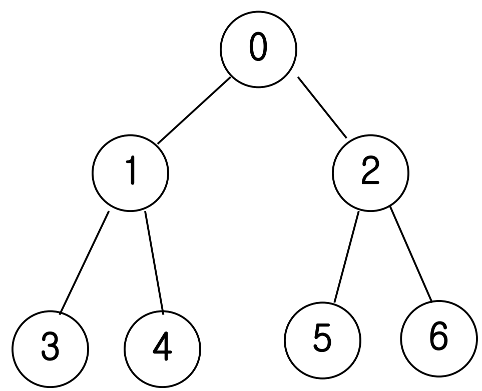
- 방향 그래프(diredted graph) : 각 간선은 방향을 가짐. <v_0,v_1>에서 v_0는 꼬리(tail)이고 v_1은 머리(head)이다. <v_0,v_1>과 <v_1,v_0>은 서로 다른 간선임.

샘플 그래프

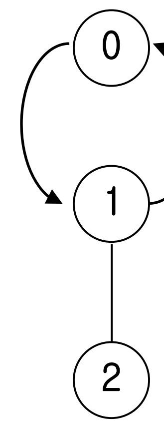
- $V(G_1) = 0,1,2,3, E(G_1) = (0,1),(0,2),(0,3),(1,2),(1,3),(2,3)$
- $V(G_2) = 0,1,2,3,4,5,6, E(G_2) = (0,1),(0,2),(1,3),(1,4),(2,5),(2,6)$
- $V(G_3) = 0,1,2, E(G_1) = <0,1>,<1,0>,<1,2>$



G_1



G_2



G_3

그래프 용어

- 완전 그래프(complete graph) : 최대수의 간선을 가지는 그래프
  (n개의 정점을 가지는 무방향 그래프의 최대 간선수는 $i \neq j$인 서로다른 무순서쌍 (v_i,v_j)의 수로 $n(n-1)/2$이다.
  (정점의수가 n인 방향 그래프에서는 $n(n-1)$개 이다.)
- 부분 그래프(subgraph) : $v(G1) \subseteq v(G)$이고 $E(G1) \subseteq E(G)$인 그래프 G1
- 그래프 G에서 정점 v_p로 부터 정점 v_q까지의 경로(Path) : 간선들 (v_p,v_{i1}), $(v_{i1},v_{i2}),\ldots,(v_{in},v_q)$에서 정점들 v_p, v_{i1}, $v_{i2},\ldots$,v_{in},v_q를 말한다.
- 경로의 길이 : 경로상에 있는 간선들의 수
- 단순 경로(simple path) : 한 경로상에 있는 모든 정점들이 다를 경우
- 단순 방향 경로(simple directed path)
- 사이클(cycle) : 처음과 마지막 정점이 같은 단순 경로

그래프 용어

- 차수(degree) : 그 정점에 부속한 간선들의 수
- 방향 그래프에서 진입 차수(in-degree), 진출 차수(out-degree)
- Self Loop : 임의의 정점 i에서 자기 자신으로 이어지는 간선.(v_I,v_I)
- 다중그래프(multigraph) : 같은 간선을 중복해서 가지는 경우
- 연결 요소(connected componet) : 최대 연결 부분 그래프(maximal connected subgraph)

그래프의 추상 데이터 타입

Structure Graph

object : 공집합이 아닌 정점의 집합과 무방향 간선의 집합으로 각 간선의 정점의 쌍으로 구성됨.

functions :

모든 graph ∈ Graph, **v**, v_1, v_2 ∈ Vertices

Graph InsertVertex(graph, v)

Graph InsertEdge(graph, v_1, v_2)

Graph DeleteVertex(graph,v)

Graph DeleteEdge(graph, v_1,v_2)

Boolean IsEmpty(graph)

List Adjacent(graph,v)

end Graph

그래프의 표현법

- 인접 행렬(adjacendy matrices)
- 인접 리스트(adjacency lists)
- 인접 다중 리스트(adjacency multilists)

인접 행렬 표기법

- G = (V,E)를 정점의 수가 n인 그래프인 경우, G의 인접행렬은 n*n의 이차원 배열
- 간선 (vi,vj)가 E(G)에 속하면 인접 행렬[i][j] = 1, 아니면 = 0
- 그림 6.7
- 무방향 그래프에서 어떤 정점 i의 차수는 그 행의 합인

$$\sum_{j=0}^{n-1} adj\_mat[i][j]$$

이다.

- 방향 그래프에서 행의 합은 진출차수이고 열의 합은 진입 차수이다.

인접 리스트 표기법

- 인접 행렬의 n행들을 n개의 연결 리스트로 표현
- N개의 정점과 e개의 간선을 갖는 무방향 그래프는 n개의 헤드 노드와 2e개의 리스트노드를 필요로 한다.

```
typedef struct node *node_pointer;
typedef struct node {
        int vertex;
        struct node *link;
        };
node_pointer graph[MAX_VERTICES];
```

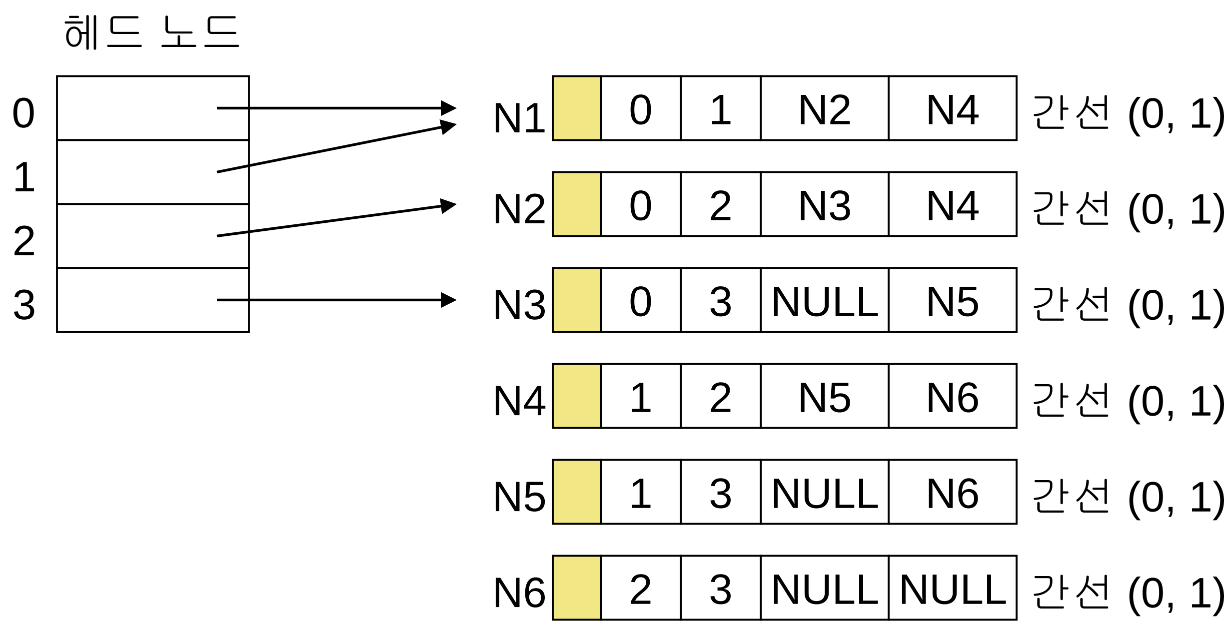
인접 다중 리스트 표기법

- 인접 리스트의 문제점 : 하나의 간선이 두 리스트에 나타남.
- 노드를 공유

marked	vertex1	vertex2	path1	path2

```
typedef struct edge *edge_pointer;
typedef struct edge {
        short itn marked;
        int vertex1;
        int vertex2;
        edge_pointer path1;
        edge_pointer path2;
        };
edge_pointer graph[MAX_VERTICES];
```

G_1 그래프 에 대한 인접 다중리스트



리스트 : 정점 0 : N1 → N2 → N3

정점 1 : N1 → N4 → N5

정점 2 : N2 → N4 → N6

정점 3 : N3 → N5 → N6

Program List

리스트의 앞에 단순 삽입

```
void insert(list_pointer *ptr, list_pointer node)
{
/* data =50인 새로운 노드를 리스트 ptr의 node 뒤에 삽입 */
	list_pointer temp;
	temp = (list_pointer)malloc(sizeof(list_node));
	if (IS_FULL(temp)) {
		fprintf(stderr, "The memory is full₩n");
		exit(1);
	}
	temp->data = 50;
	if (*ptr) {
		temp->link = node->link;
		node->link = temp;
	}
	else {
		temp->link = NULL;
		*ptr = temp;
	}
}
```

리스트 삭제, 출력

```
void delete(list_pointer *ptr, list_pointer trail, list_pointer node)
{
/* 리스트로부터 노드를 삭제, trail은 삭제될 node의 선행 노드이며
ptr은 리스트의 시작 */
        if (trail)
                trail->link = node->link;
        else
                *ptr = (*ptr)->link
        free(node);
}


void print_list(list_pointer ptr)
{
        printf(“The list contains: “);
        for(; ptr; ptr = ptr->link)
                printf(“%4d, ptr->data);
                printf(“₩n”);
}
```

연결된 스택에서의 삽입

```
void add(stack_pointer *top, element item)
{
/* 스택의 톱에 원소를 삽입 */
	stack_pointer temp = (stack_pointer)malloc(sizeof(stack));
	if (IS_FULL(temp)) {
		fprintf(stderr, "The memory is full₩n");
		exit(1);
	}
	temp->item = item;
	temp->link = *top;
	*top = temp;
```

연결된 스택에서의 삭제

```
element delete(stack_pointer *top) {
/* 스택으로부터 원소를 삭제 */
        stack_pointer temp = *top;
        element item;
        if (IS_EMPTY(temp)) {
                fprintf(stderr, "The stack is empty₩n");
                exit(1);
        }
        item = temp->item;
        *top = temp->link;

        free(temp);
        return item;
}
```

연결된 큐의 rear에 삽입

```
void addq(queue_pointer *front, queue_pointer *rear, element item)
{
/* 큐의 rear에 원소를 삽입 */
        queue_pointer temp = (queue_pointer)malloc(sizeof(queue));
        if (IS_FULL(temp)) {
                fprintf(stderr, “The memory is full₩n”);
                exit(1);
        }
        temp->item = item;
        temp->link = NULL;
        if (*front)(*rear)->link = temp;
        else *front = temp;
        *rear = temp;
}
```

연결된 큐의 앞으로부터 삭제

```
element deleteq(queue_pointer *front) {
/* 큐에서 원소를 삭제 */
{
        queue_pointer temp = *front;
        element item;
        if (IS_EMPTY(temp)) {
                fprintf(stderr, “The queue is empty₩n”);
                exit(1);
        }
        item = temp->item;
        *front = temp->link;
        free(temp);
        return item;
}
```

이중 연결 원형 리스트에 삽입, 삭제

```
void dinsert(node_pointer node, node_pointer newnode)
{
/* newnode를 node의 오른쪽에 삽입 */
        newnode->llink = node;
        newnode->rlink = node->rlink;
        node->rlink->llink = newnode;
        node->rlink = newnode;
}


void ddelete(node_pointer node, node_pointer deleted)
{
/* 이중 연결 리스트에서 삭제 */
        if (node == deleted)
                printf("Deletion of head node not permitted.₩n");
        else {
                deleted->llink->rlink = deleted->rlink;
                deleted->rlink->llink = deleted->llink;
                free(deleted);
        }
}
```

이진 탐색 트리의 반복적 탐색

```
tree_pointer search2(tree_pointer tree, int key)
{
/* 키값이 key인 노드에 대한 포인터를 반환함. 그런 노드가 없는 경우는
NULL을 반환 */
        while (tree) {
                if (key == tree->data) return tree;
                if (key < tree->data)
                                tree = tree->left_child;
                else
                                tree = tree->right_child;
        return NULL;
}
```

이진 탐색 트리에 원소를 삽입

```
void insert_node(tree_pointer *node, int num)
/* 트리내의 노드가 num을 가리키고 있으면 아무일도 하지 않음.
그렇지 않은 경우는 data=num인 새 노드를 첨가 */
{
	tree_pointer ptr, temp = modified_search(*node, num);
	if (temp || !(*node)) {
		/* num이 트리내에 없음 */
		ptr = (tree_pointer) malloc(sizeof(node));
		if (IS_FULL(ptr)) {
			fprintf(stderr, “The memory is full₩n”);
			exit(1);
		}
		ptr->data = num;
		ptr->left_child = ptr->right_child = NULL;
		if (*node)  /* temp의 자식으로 삽입 */
			if (num < temp->data) temp->left_child = ptr;
			else temp->right_child = ptr;
		else *node = ptr;
	}
}
```

히프 트리

```
void heapsort(element list[], int n)
/* 배열에 대한 히프 정렬을 수행하는 알고리즘 */
{
        int i, j;
        element temp;

        for (i = n/2; i > 0; i--)
                adjust(list, i, n);
        for (i = n-1; i > 0; i--)
                SWAP(list[1], list[i+1], temp);
                adjust(list, 1, i);
        }
}
```

최대 히프의 조정

```
void adjust(element list[], int root, int n)
/* 히프를 구성하기 위하여 이진 트리를 조정하는 알고리즘 */
{
    int child, rootkey;
    element temp;
    temp = list[root];
    rootkey = list[root].key;
    child = 2*root;   /* 왼쪽 자식 */
    while (child <= n) {
        if ((child < n) && (list[child].key < list[child+1].key))
                child ++;
        if (rootkey > list[child].key)   /* 부모와 최대값의 자식과 비교 */
                break;
        else {
                list[child/2] = list[child]; /* 부모로 이동 */
            child *= 2;
        }
    }
    list[child/2] = temp;
}
```