- **정렬 (Sorting) 알고리즘**

데이터를 크기 순으로 재배치하는 알고리즘. 대부분의 정렬 알고리즘은 $O(n^2)$과 $O(n \log n)$ 사이의 시간이 소요됨.

- **선택 정렬 (Selection Sort)**

각 루프마다 최대 원소를 찾는다.
최대 원소와 맨 오른쪽 원소를 교환한다.
맨 오른쪽 원소를 제외한다.
하나의 원소만 남을 때까지 위의 루프를 반복.

Init	29	10	14	**37**	13
1st	**29**	10	14	13	37
2nd	13	10	**14**	29	37
3rd	**13**	10	14	29	37
4th	10	13	14	29	37

```
01. selectionSort(A[], n){
02.   for last = n down to 2 {
03.     A[k] = max(A[1...last]);
04.     swap(A[k], A[last]);
05.   }
06. }
```

2번 줄의 반복문은 n-1번 반복.

3번 줄의 max 함수의 비교횟수: last-1

4번 줄의 swap 함수는 상수 시간 작업.

수행시간:

$$(n-1) + (n-2) + \cdots + 2 + 1 = O(n^2)$$

- **버블 정렬 (Bubble Sort)**

최악의 경우 매 pass마다 모든 데이터들의 자리 교환이 발생.

Init	29	10	14	37	13
1st	**29**	**10**	14	37	13
	10	**29**	**14**	37	13
	10	14	29	**37**	**13**
	10	14	29	13	37
2nd	10	14	**29**	**13**	37
	10	14	13	29	37
3rd	10	**14**	**13**	29	37
	10	13	14	29	37

```
01. bubbleSort(A[], n){
02.   for last = n down to 2
03.     for i = 1 to last – 1
04.       if(A[i] > A[i+1]) then
05.         swap(A[i], A[i+1]);
06. }
```

2번 줄의 반복문은 n-1번 반복.

3번 줄의 반복문은 n-1, n-2, … , 2, 1번 반복.

4, 5번 줄은 상수 시간 작업.

수행시간:

$$(n-1) + (n-2) + \cdots + 2 + 1 = O(n^2)$$

- **삽입 정렬 (Insertion Sort)**

매 i번째 단계에서 i+1개의 데이터를 정렬한다.

Init	29	10	14	37	13
1st	29	**10**	14	37	13
	29	29	14	37	13
	10	29	14	37	13
2nd	10	29	**14**	37	13
	10	29	29	37	13
	10	14	29	37	13
3rd	10	14	29	**37**	13
	10	14	29	37	13
4th	10	14	29	37	**13**
	10	14	14	29	37
	10	13	14	29	37

```
01. insertionSort(A[], n){
02.   for i = 2 to n
03.     insert(A[i], A[1...i]);
04. }
```

2번 줄의 반복문 n-1번 반복

3번 줄의 삽입은 최악의 경우 i-1회 비교

수행시간 Worst case:

$$1+2+\cdots+(n-2)+(n-1)=\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}$$
$$=O(n^2)$$

수행시간 Average case:

$$\frac{1}{2}\big(1+2+\cdots+(n-2)+(n-1)\big)$$
$$=\frac{1}{2}\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}=O(n^2)$$

- **선택, 버블 vs 삽입**

선택, 버블 정렬은 n개의 배열에서 시작하여 정렬해야 하는 배열의 크기를 하나씩 줄여 나감.

삽입 정렬은 1개의 배열에서 시작하여 이미 정렬한 배열의 크기를 하나씩 늘려 나감.

- **병합 정렬 (Merge Sort)**

Init	31	3	65	73	8	11	20	29	48	15
1st	31	3	65	73	8	11	20	29	48	15
2nd	3	31	65	73	8	11	20	29	15	48
3rd	3	8	11	20	29	31	65	73	15	48
4th	3	8	11	15	20	29	31	48	65	73

2nd	**3**	31	65	73	8	11	20	29
		31	65	73	**8**	11	20	29
	3							
		31	65	73		**11**	20	29
	3	8						
		31	65	73			**20**	29
	3	8	11					
		31	65	73				**29**
	3	8	11	20				
		31	65	73				
	3	8	11	20	29			
	3	8	11	20	29	31	65	73

```
01. mergeSort(A[], p, r){
02.   if(p < r) then {
03.     q = floor((p + r) / 2);
04.     mergeSort(A, p, q);
05.     mergeSort(A, q + 1, p);
06.     merge(A, p, q, r);
07.   }
08. }
09.
10. merge(A[], p, q, r){
11.   sort(A[p...q], A[q+1...r]);
12. }
```

수행시간:

$$T(n)=2T(n/2)+cn=2^2T(n/2^2)+2cn$$
$$=2^3T(n/2^3)+3cn=\cdots$$
$$=2^kT(n/2^2)+kcn \xrightarrow{n=2^k} n+cn\log n$$
$$=O(n\log n)$$

- **퀵 정렬 (Quick Sort)**

Init	31	3	65	73	8	11	20	29	48	**15**
1st	8	11	**3**	15	31	48	20	29	65	**73**
2nd	3	11	**8**	15	31	48	20	29	**65**	73
3rd	3	8	**11**	15	31	48	20	**29**	65	73
4th	3	8	11	15	**20**	29	31	**48**	65	73
5th	3	8	11	15	20	29	**31**	48	65	73
6th	3	8	11	15	20	29	31	48	65	73

Init	31	3	65	73	8	11	20	29	48	**15**
	31	3	65	73	8	11	20	29	48	**15**
	3	31	65	73	8	11	20	29	48	**15**
	3	31	65	73	8	11	20	29	48	**15**
	3	31	65	73	8	11	20	29	48	**15**
	3	8	65	73	31	11	20	29	48	**15**
	3	8	11	73	31	65	20	29	48	**15**
	3	8	11	73	31	65	20	29	48	**15**
	3	8	11	73	31	65	20	29	48	**15**
	3	8	11	73	31	65	20	29	48	**15**
	3	8	11	15	31	65	20	29	48	73

```
01. quicksort(A[], p, r){
02.   if(p < r) then {
03.     q = partition(A, p, r);
04.     quickSort(A, p, q - 1);
05.     quicksort(A, p + 1; r);
06.   }
07. }
08.
09. partition(A[], p r){
10.   x = A[r]; i = p -1;
11.   for j = p to r – 1
12.     if(A[j] == x) then
13.       swap(A[++i], A[j]);
14.   swap(A[i + 1], A[r]);
15.   return i + 1;
16. }
```

수행시간 Worst case: $O(n^2)$

수행시간 Average case: $O(n \log n)$

- **힙 정렬 (Heap Sort)**

Heap은 완전 이진 트리로, 다음의 성질을 만족한다

각 부모 노드는 자신의 자식 노드의 값보다
작거나 같으면 최소힙, 크거나 같으면 최대힙.

내림차순으로 정렬하려면 최대힙을,
오름차순으로 정렬하려면 최소힙을 구성한다.

```
01. heapsort(A[], n){
02.   buildHeap(A, n);
03.   for i = n down to 2{
04.     swap(A[1], A[i];
05.     heapify(A, 1, i - 1);
06.   }
07. }
```

수행시간 Worst case:

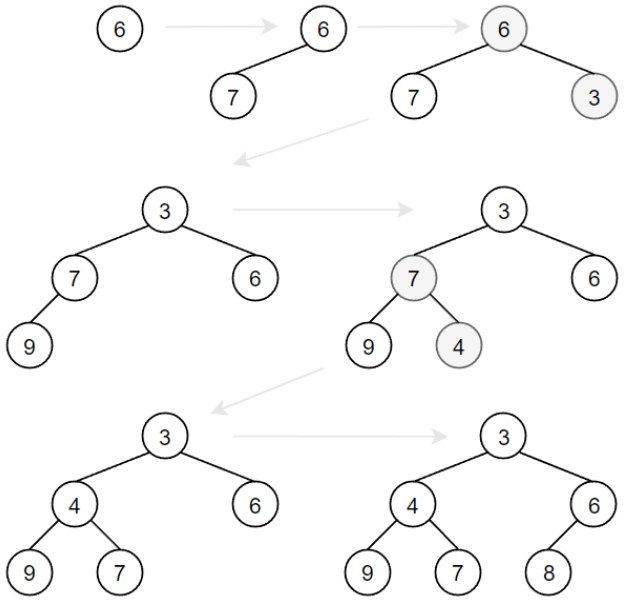
$$2^1 \times 1 + 2^2 \times 2 + \cdots + 2^k \times k$$
$$< (2^1 + 2^2 + \cdots + 2^k) \times k$$
$$= (2^{k+1} - 2\ ) \times k \approx O(n \log n)$$

수행시간 Average case:

$$O(n)$$

- **Build Heap**

6	7	3	9	4	8



- **각 정렬 알고리즘의 효율성**

	Worst case	Average case
선택	$O(n^2)$	$O(n^2)$
버블	$O(n^2)$	$O(n^2)$
삽입	$O(n^2)$	$O(n^2)$
병합	$O(n \log n)$	$O(n \log n)$
퀵	$O(n^2)$	$O(n \log n)$
힙	$O(n \log n)$	$O(n \log n)$

- **Quick Select**

2번째 작은 원소 찾기

Init	31	3	65	73	8	11	20	29	48	**15**
1st	8	11	**3**	15	31	48	20	29	65	73
2nd	3	8	**11**							
3rd		**8**	11							
4th		8								

```
01. select(A[], p, r, i){
02.   if(p == r) then
03.     return A[p];
04.
05.   q = partition(A, p, r);
06.   k = q – p + 1;
07.
08.   if(i < k) then
09.     return select(A, p, q - 1, i);
10.   else if(i == k) then
11.     return A[q]
12.   else
13.     return select(A, q + 1, r, i - k);
14. }
```

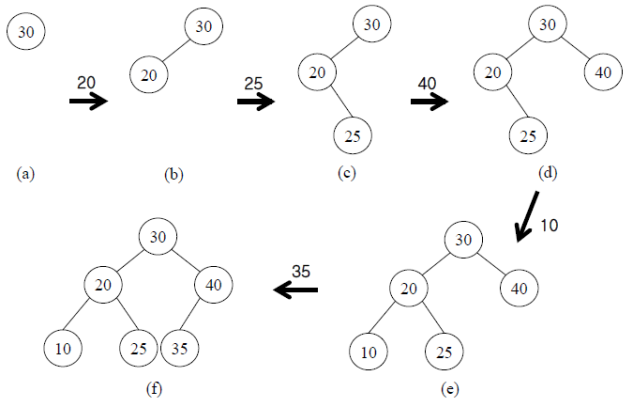
수행시간 Worst case: $O(n^2)$

수행시간 Average case: $O(n)$

- **이진 검색 트리 (Binary Search Tress, BST)**

```
01. search(t, x){
02.   if(t == NULL or key[t] == x) then
03.     return t;
04.   if(x < key[t]) then
05.     return search(left[t], x);
06.   else return search(right[t], x);
07. }
```

- **BST에서의 삽입 알고리즘**

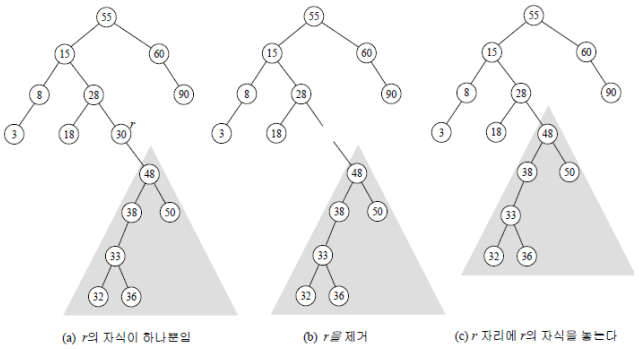



```
01. insert(t, x){
02.   if(t == NULL) then {
03.     key[t] = x;
04.     left[t] = NULL;
05.     right[t] = NULL;
06.     return t;
07.   }
08.
09.   if(x < key[t]) then
10.     return insert(left[t], x);
11.   else return insert(right[t], x);
12. }
```

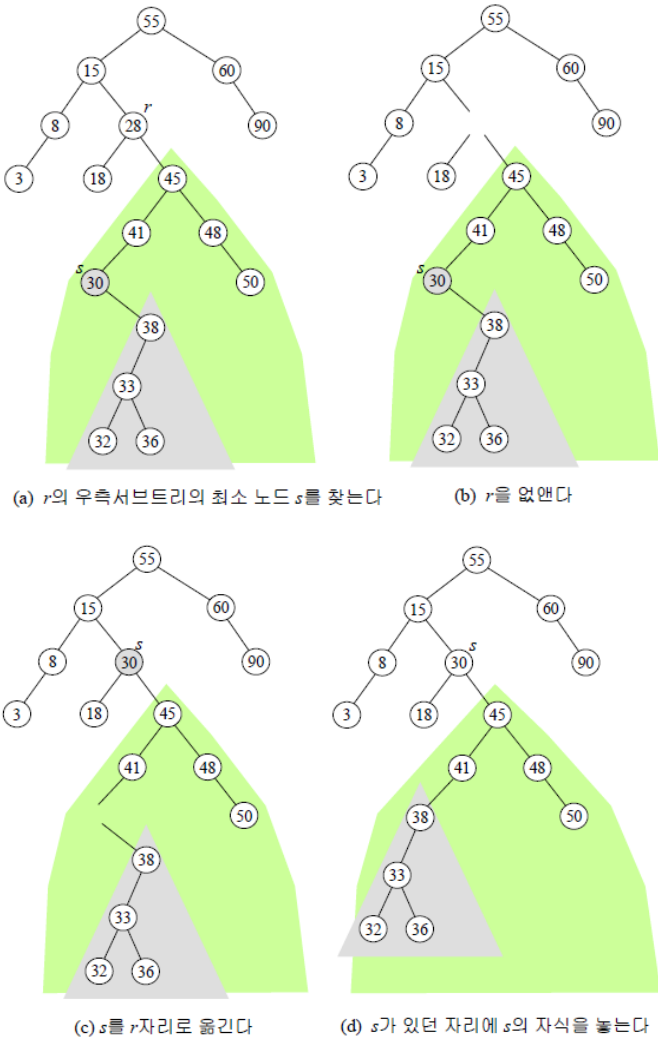
- **BST에서의 삭제 알고리즘**

```
01. delete(t, r, p){
02.   if(r == t) then root = dropNode(r);
03.   else if(r == left[p]) then
04.     left[p] = dropNode(r);
05.   else right[t] = dropNode(r);
06. }
07.
08. dropNode(r){
09.   if(left[r] == right[r] == NULL) then
10.     return NULL;
11.   else if(left[r] == NULL and
12.           right[r] != NULL) then
13.     return right[r];
14.   else if(left[r] != NULL and
15.           Right[r] == NULL) then
16.     return left[r];
17.   else {
18.     s = left[r];
19.     while(right[s] != NULL){
20.       parent = s;
21.       s = right[s];
22.     }
23.     key[r] = key[s];
24.     if(s == left[r]) then left[r] = left[s]
25.     else right[parent] = left[s];
26.     return r;
27.   }
28. }
```

- **BST에서 삭제알고리즘의 예#1**



- **BST에서 삭제 알고리즘의 예#2**



- **BST의 검색 시간 복잡도**

Worst case: $O(n)$

Average case: $O(\log n)$

- **B-Tree**

대표적인 다진 검색 트리(Multiway Search Tree). 다진 검색 트리는 자식 노드의 수가 2개를 초과한다.

하나의 노드에 저장되는 key의 개수는

$$\text{number of child node} - 1$$

즉, k개의 key가 저장되는 노드의 자식 노드 수는

$$k + 1$$

B-Tree는 균형 잡힌 다진 검색 트리로, 다음의 성질을 만족한다.

■ 하나의 노드가 최대 k 개의 key를 가질 수 있을 때, 루트를 제외한 모든 노드는 $\lfloor k/2 \rfloor$~k 개의 키를 갖는다.

■ 노드의 key 수가 $\lfloor k/2 \rfloor$ 개보다 작으면 underflow.

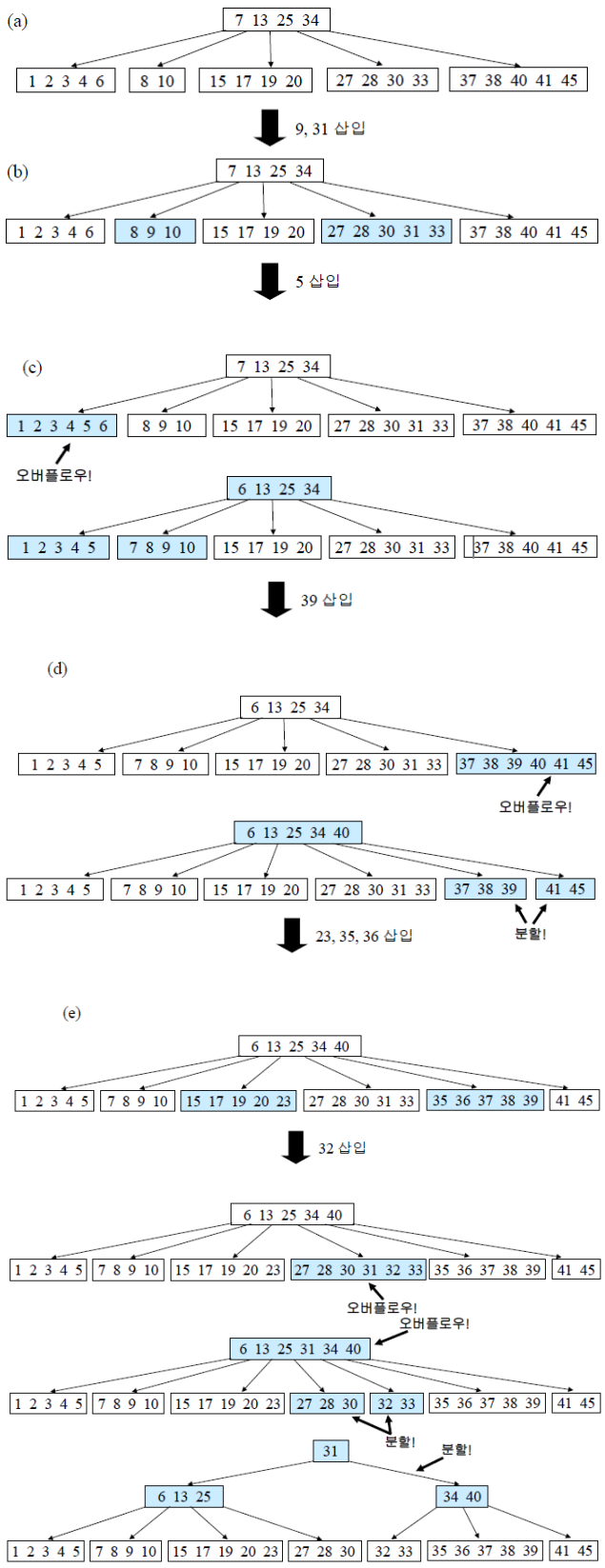
■ 노드의 key 수가 $\lfloor k/2 \rfloor$ 개보다 크면 overflow.

■ 모든 리프 노드는 깊이가 같다.

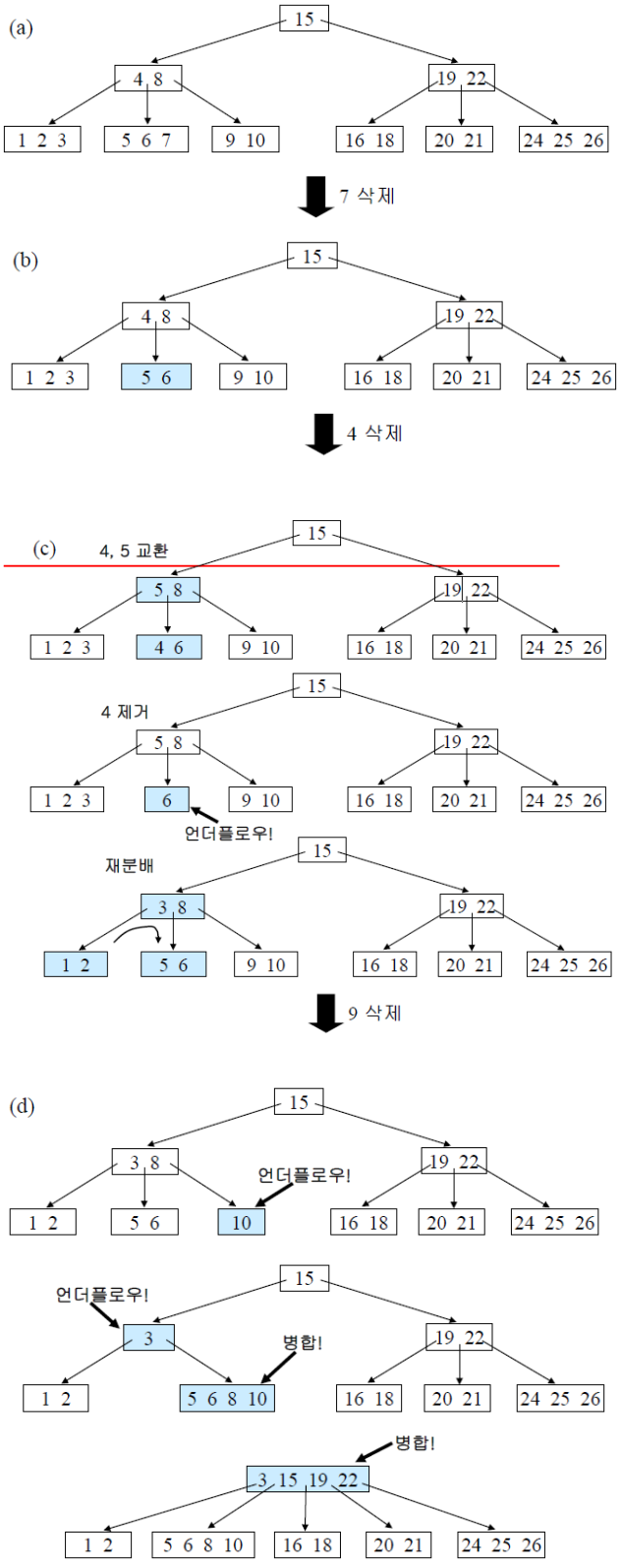
- **B-Tree 에서의 삽입 알고리즘**

```
01. insert(t, x){
02.   r = proper leaf[t] to insert x;
03.   insert x to r;
04.   if(isOverflow(r)) then
05.     clearOverflow(r);
06. }
07.
08. clearOverflow(r){
09.   s = siblings(r) that not full;
10.   p = parent[r];
11.   if(s != NULL) then {
12.     give proper edgeOf[r] to p;
13.     give proper oneOf[r] to s
14.   } else {
15.     c = middleOf[r];
16.     p = parent[r];
17.     split(r);
18.     give c to p;
19.     if(isOverflow(p) then
20.       clearOverflow(p);
21.   }
22. }
```

- **B-Tree에서 삽입의 예**



- **B-Tree에서 삭제의 예**



- **해시 테이블 (Hash Table)**

  원소가 저장될 자리가 원소의 값에 의해 결정되는 자료구조. 평균 상수 시간에 삽입, 삭제, 검색이 가능. Hash table은 최대 최소 원소 또는 i번째 작은 큰 원소를 찾는 것과 같은 작업은 지원하지 않는다.

- **Hash Function**

  입력 원소가 Hash Table에 고르게 저장되는 함수가 바람직하며, 계산이 간단하고 충돌이 적어야 한다. 가장 대표적인 것은 Division Method다.

  입력 원소: 25, 13, 16, 15, 7
  해시 함수: $h(x) = x \bmod 13$

0	13	7	7
1		8	
2	15	9	
3	16	10	
4		11	
5		12	25
6			

- **Division Method Hash Function**

$$h(x) = x \bmod m$$

  m은 테이블 사이즈이며 보통 짝수보다는 홀수, 홀수 보다는 소수(prime number)를 사용한다.

  m이 짝수이면, 입력값이 짝수 또는 홀수로 편중될 경우 해시값이 편향될 수 있음.

- **Multiplication Method Hash Function**

$$h(x) = \lfloor (xA \bmod 1)m \rfloor$$

  A는 $0 < A < 1$인 상수이다. 이 경우 m은 굳이 소수가 아니어도 되므로 보통 2^p으로 한다.

- **Collision**

  Hash Table의 한 주소를 놓고 두 개 이상의 원소가 자리를 다투는 상황이다. Hashing을 해서 삽입할 때 이미 다른 원소가 자리를 차지하고 있는 상황.

  Collision을 해결하는 방법은 크게 두가지 있다.
  →Chaining, Open Addressing

- **Collision Resolution: Chaining**

  같은 주소로 hashing되는 원소를 모두 하나의 linked list로 관리한다.

  단점: 추가적인 자료 구조인 linked list가 요구된다.

- **Collision Resolution: Open Addressing (개방 주소)**

  Collision이 일어나더라도 어떻게든 주어진 테이블 공간에서 해결한다. 따라서 추가적인 공간이 필요하지 않다. 다음 세가지 방법이 있다.

  선형 조사 (Linear Probing)
  이차원 조사 (Quadratic Proving)
  더블 해싱 (Double Hashing)

- **Open Addressing: 선형 조사 (Linear Probing)**

  가장 간단한 방법으로 아래의 함수를 따른다.

$$h_i(x) = (h(x) + i) \bmod m, h(x) = x \bmod m$$

  입력 순서: 25, 13, 16, 15, 7, 28, 31, 20, 1, 38
  해시 함수: $h_i(x) = (h(x) + i) \bmod 13$

0	13	0	13	0	13
1		1		1	1
2	15	2	15	2	15
3	16	3	16	3	16
4	**28**	4	28	4	28
5		5	31	5	31
6		6		6	**38**
7	7	7	7	7	7
8		8	**20**	8	20
9		9		9	
10		10		10	
11		11		11	
12	25	12	25	12	25

  단점: 특정 영역에 원소가 집중하게 되는 경우에 취약하다. 여러 번의 재해싱이 요구되기 때문.

- **Open Addressing: 이차원 조사 (Quadratic Proving)**

$$h_i(x) = (h(x) + c_1 i^2 + c_2 i) \bmod m$$

  입력 순서: 15, 18, 43, 37, 45, 30
  해시 함수: $h_i(x) = (h(x) + i^2) \bmod m$

0		5	18	10	
1		6	45	11	37
2	15	7		12	
3		8	**30**		
4	43	9			

- **Open Addressing: 더블 해싱 (Double Hashing)**

  두개의 해싱 함수를 사용.

  $$h_i(x) = \big(h(x) + i \cdot xf(x)\big) \bmod m$$

  입력 순서: 15, 19, 28, 41, 67

  해시 함수: $h(x) = x \bmod 13$

  해시 함수: $h(x) = (x \bmod 11) + 1$

0		5		10	
1		6	19	11	**41**
2	15	7		12	
3		8			
4	**64**	9	**28**		

- **Hash Table에서의 검색 시간**

  평균 검색 시간 O(1)이 소요됨. 해싱 방법에 따라 소요 시간이 다를 수 있다.

  검색 시간은 Load factor (적재율; α)에 영향을 받는다. Load factor는 Hash Table 전체에서 얼마나 원소가 저장되어 있는지를 나타내는 값이다.

  $$\alpha = \frac{n}{m}$$

  Chaining 기반 Hash Table의 적재율이 1을 초과하면, Linked list의 검색 시간이 초과된다.

  적재율이 높아지면 Hash Table의 효율이 떨어진다. 일반적으로, 적정선(threshold)을 미리 설정해 놓고 적재율이 적정선에 이르면 Hash Table의 크기를 증가 후 저장되어 있는 모든 원소를 다시 hashing.

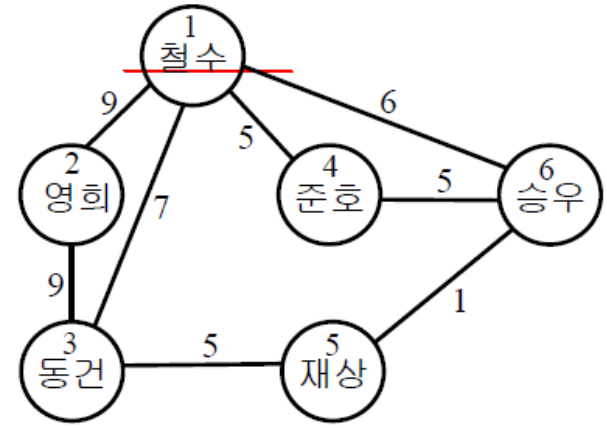
- **그래프(Graph)**는 현상이나 사물을 정점(Vertex)와 간선(edge)로 표현한 것이다.

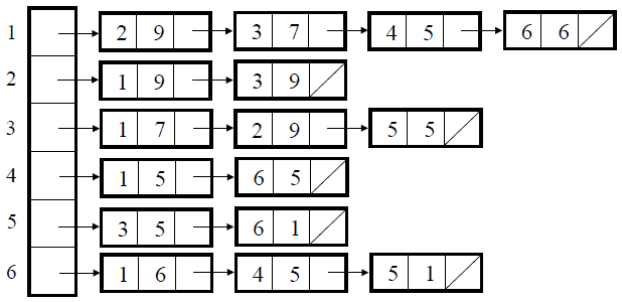
  Graph $G = (V, E)$, V는 정점 집합, E는 간선 집합니다. 두 정점이 간선으로 연결되어 있으면 인접(adjacent)해 있다고 한다.

- **Adjacency Matrix**는 $N \times N$ 행렬로 표현한다.

- **Adjacency List**는 각 정점에 인접한 정점들을 리스트로 표현한다.

- **Graph 표현의 예**

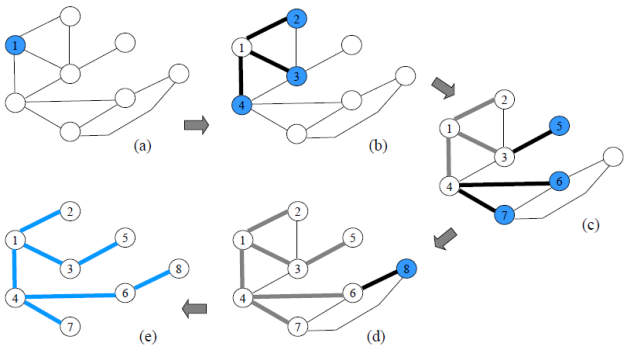




- **Graph Traversal**의 대표적인 두 가지 방법 BFS, DFS

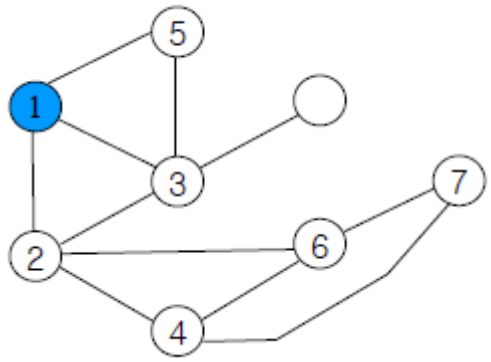
- **너비 우선 탐색 (BFS)**

```
01. BFS(G, s){
02.   for each v in V - {s}
03.     visited[v] = NO;
04.   visited[s] = YES;
05.   enqueue(Q, s);
06.   while(!isEmpty(Q)){
07.     u = dequeue(Q);
08.     for each v in neighbors(u)
09.       if(visited[v] = NO) then {
10.         visited[v] = YES;
11.         enqueue(Q, v);
12.       }
13.   }
14. }
```



- **깊이 우선 탐색 (DFS)**

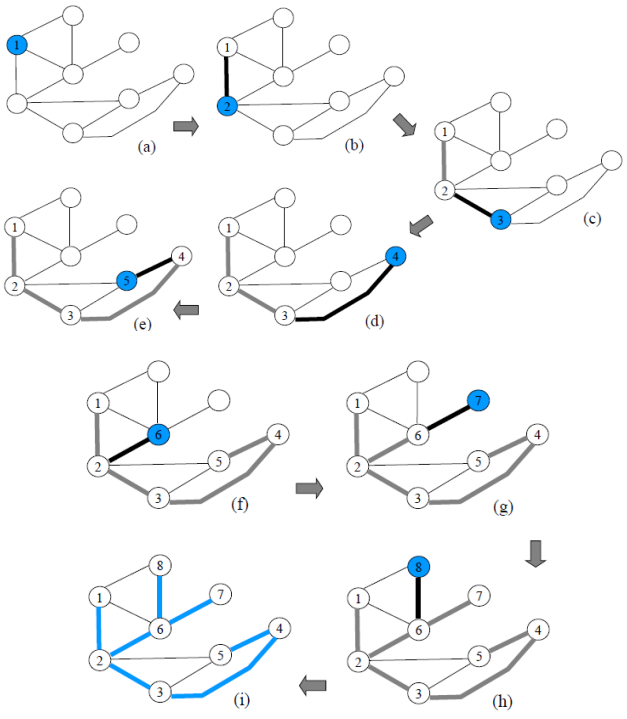
```
01. DFS(G){
02.   for each v in V
03.     visited[v] = NO;
04.   for each v in V
05.     if(visited[v] == NO) visit(v);
06. }
07.
08. visit(v){
09.   visited[v] = YES;
10.   for each x in neighbors(v)
11.     if(visited[x] == NO) then visit(x);
12. }
```



11번 줄에서 위와 같은 문제가 생긴다. 2번 vertex에서 3, 4 모두 인접 vertex이므로, 3으로 가면 BFS와 같게 된다.

이는 인접 리스트를 왼쪽 자식에서 오른쪽 자식 순으로 구성하면 해결된다. 즉, 인접 리스트를 다음과 같이 구성한다.

{1 | 2, 3, 4}, {2 | 4, 6, 3}, {4 | 7, 6, 2}



- **최소 신장 트리 (Minimum Spanning Trees)**

최소 신장 트리는 무향 연결 그래프이며, Cycle이 없는 그래프이다.

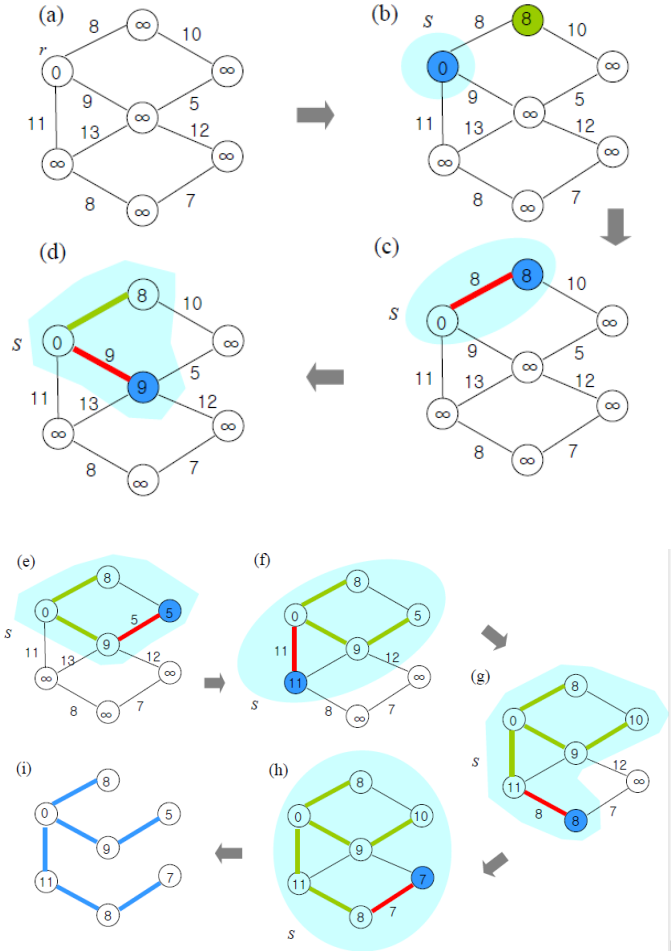
무향 연결 그래프는 모든 정점 간에 경로가 존재하는 그래프이다. Cycle이 없는 연결 그래프는 트리의 조건이며, n개의 정점을 갖는 트리는 항상 n-1개의 간선을 갖는다.

최소 신장 트리는 그래프 G의 신장 트리들 중 간선의 가중치의 합이 최소인 신장트리이다.

그래프를 최소 신장 트리로 만드는 대표적인 알고리즘은 Prim 알고리즘과 Kruskal 알고리즘이 있다.

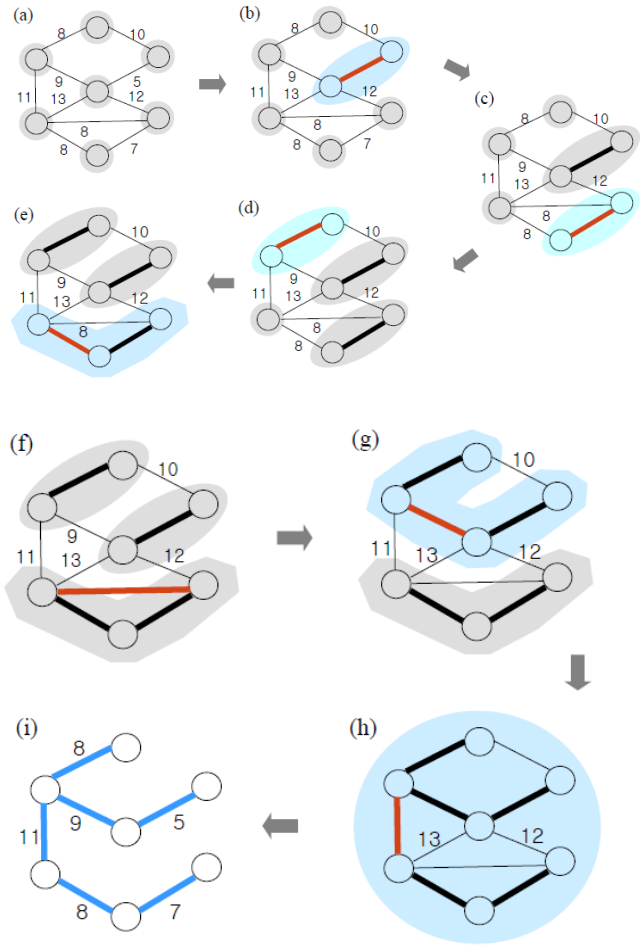
- **Prim Algorithm**

```
01. Prim(G, r){
02.   S = {};
03.   visited[r] = YES;
04.   S = S cup r
05.   while(S != V){
06.     (x in S, y in V - S)
07.       = min(edges(S, V - S));
08.     visited[y] = YES;
09.     S = S cup y;
10.   }
11. }
```



- **Kruskal Algorithm**

```
01. Kruskal(G){
02.   T = {}; // spanning tree
03.   A = sort as ascend E;
04.   M = 0;
05.
06.   while(M <n - 1){
07.     (u in U, v in V)
08.       = pop_front(A); // min edge
09.     M++;
10.     if(U != V){
11.       U = U cup V;
12.       T = T cup (u, v);
13.     }
14.   }
15. }
```



- **최단 경로 (Shortest Paths) 알고리즘**

그래프 내의 한 정점에서 다른 정점으로 이동할 때 가중치의 합이 최소값이 되도록 만드는 경로를 찾는 알고리즘.

조건은, 가중치가 있는 유향 그래프 여야 한다.

두 정점 간의 최단 경로를 찾는 알고리즘에는 Dijkstra 알고리즘과 Bellman-Ford 알고리즘이 있다. 다익스트라 알고리즘은 음의 가중치가 없는 그래프에서 최단 경로이며, 벨만-포드는 음의 가중치를 허용한다.

모든 정점 간의 최단 경로를 찾는 알고리즘에는 Floyd-Warshall (플로이드-워샬) 알고리즘이 있다.

- **Dijkstra Algorithm**

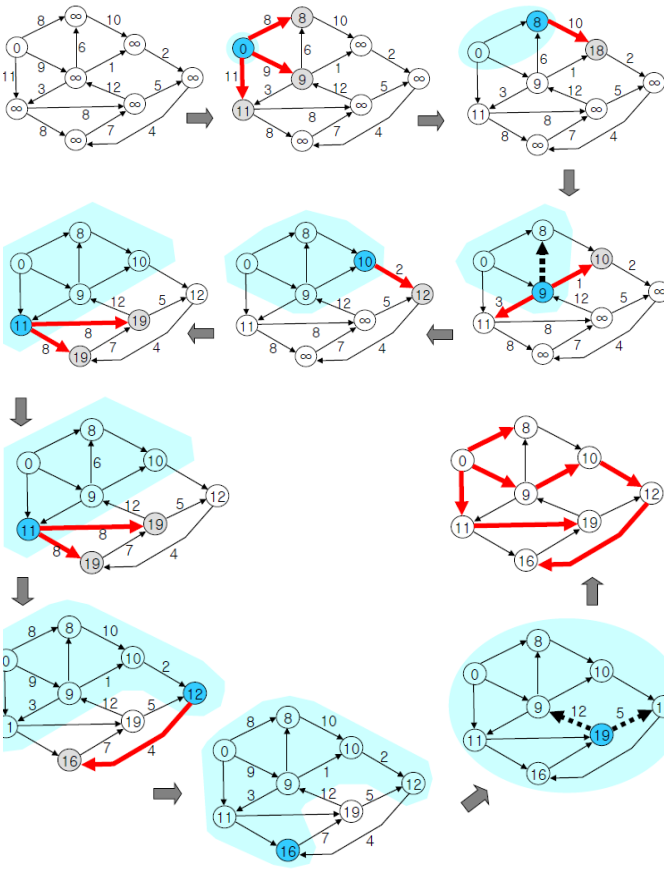
프림 알고리즘이 단순히 간선의 길이를 이용해 어떤 간선을 먼저 연결할지를 결정하는데 반해, 다익스트라 알고리즘은 경로의 길이를 감안해서 간선을 연결한다.

1. 각 정점 위에 시작점으로부터 자신에게 이르는 경로의 길이를 저장할 곳을 준비하고 모든 정점 위에 있는 경로의 길이를 무한대로 초기화 한다.
2. 시작 정점의 경로 길이를 0으로 초기화하고 최단 경로에 추가한다.
3. 최단 경로에 새로 추가된 정점의 인접 정점들에 대해 경로 길이를 갱신하고, 이들을 최단 경로에 추가한다.

   만약 추가하려는 인접 정점이 이미 최단 경로 안에 존재한다면, 갱신되기 이전의 경로 길이가 새로운 경로의 길이보다 더 큰 경우에 한해, 다른 선행 정점을 지나던 기존의 경로를 현재 정점을 경유하도록 수정.
4. 그래프 내의 모든 정점이 최단 경로에 소속될 때까지 3 과정을 반복.

- **Dijkstra Algorithm**

```
01. Dijkstra(G, r){
02.   S = {};
03.   for each u in V
04.     val(u) = inf;
05.   val(r)=0;
06.   while(S != V){
07.     u = min(V-S);
08.     S = S cup u;
09.     for each v in neighbors(u)
10.       if(v in V - S and
11.          val(v) > val(u) + dist(u, v))
12.       then
13.          val(v) = val(u) + dist(u, v));
14.   }
15. }
```



- **풀 수 없는 문제들 (Unsolvable, Undecidable)**
  풀 수 있는 문제들 (Solvable, Decidable)

- **현실적인 시간, 다항식 시간 (Polynomial Time)**

  입력 크기 n에 대한 다항식으로 표시되는 시간.

  $$3n^k + 5n^{k-1} + \cdots, \Theta(n^k), \Theta(n^k \log n), \cdots$$

  n^{100}도 비현실적인 시간이지만, 현실적인 문제는 보통 n^6 정도에서 해결된다.

- **비현실적인 시간, 비다항식 시간 (Non-polynomial)**

  지수 시간 (Exponential time), 계승 시간 (Factorial)

  $$2^n, \Theta(r^n), n!, \Theta(n!)$$

- **Yes/No 문제**

  그래프 G에서, 정점 u에서 정점 v로 가는 길이 100 이하인 경로가 존재하는가?

- **최적화 문제**

  그래프 G에서, u에서 v로 가는 최단 경로의 길이는 얼마인가?

- **P (Polynomial) 문제**

  어떤 문제에 대해 yes/no 대답을 다항식 시간에 해줄 수 있는 문제. 즉, 다항식 시간에 Yes 또는 No 대답을 할 수 있으면 P 문제.

- **NP (Nondeterministic Polynomial) 문제**

  현실적인 시간에 풀 수 없는 문제 중에서 어떤 '조건'을 주었을 때 해결할 수 있는 문제.

  정의를 쉽게 하기 위해 보통 Yes/No 문제들을 대상으로 함. 어떤 문제의 대답이 yes라는 근거가 주어졌을 때 (조건), 그것이 옳은 근거임을 다항식 시간에 확인해 줄 수 있는 문제.

  Traveling Salesperson Problem (TSP)
  N개의 도시를 모두 방문하고 돌아오는 거리 K 이하인 경로가 존재하는가? cf) N개의 도시를 모두 방문하고 돌아오는 최단 거리는?

  Hamilton 문제
  어떤 그래프에서 모든 정점을 단 한번씩 방문하고 돌아오는 경로가 존재하는가?

- **NP-Hard의 이해**

  문제 1: 정수 $x = x_1 x_2 \cdots x_n$은 3의 배수인가?
  문제 2: $x_1 + x_2 + \cdots + x_n$은 3의 배수인가?

  위 두 문제의 대답은 같다. 문제 2가 쉬우면, 문제 1도 쉽다. 즉, 어떤 어려운 문제 A를, 동일한 답을 내는 보다 쉬운 문제 B로 변형이 가능하면, A도 쉬운 문제가 될 수 있다.

  문제 A는 어렵고, B는 쉽다고 하자. 문제 A가 Yes/No 대답이 일치하는 문제 B로 다항식 시간 안에 변형된다. 그렇다면, 문제 A도 쉽게 해결이 가능하게 된다.

- **NP-Hard**

  NP에 속하는 모든 문제가 다항식 시간에 변환 가능한 문제.

  만약, NP-Hard의 문제 L을 쉽게 해결하는 알고리즘이 발견된다면 모든 NP문제도 쉽게 풀 수 있다.

- **NP-Complete**

  NP이면서 NP-Hard인 문제.

  통상 현실적인 시간에 풀 수 없는 문제들 중 특정 문제들의 군. 이에 속한 문제 중 한 문제만 현실적인 시간에 풀면 다른 모든 것도 저절로 풀리는 논리적 연결관계를 갖고 있다.

  NP-Complete 이론
  NP-Complete 문제를 현실적인 시간에 풀 수 있는가에 관한 이론.

  어떤 문제가 NP-Complete/Hard임이 확인되면, 쉬운 알고리즘을 찾으려는 헛된 노력은 일단 중지한다. 주어진 시간 예산 내에서 최대한 좋은 해를 찾는 알고리즘 (heuristic) 개발에 집중한다.

-