컴퓨팅사고와 파이썬 프로그래밍

Ch 11. 파이썬 기반 자료구조와 알고리즘

교수 김 영 탁

영남대학교 정보통신공학과

(Tel : +82-53-810-2497; E-mail : ytkim@yu.ac.kr)

Outline

- 파이썬 기반 자료구조와 알고리즘
- 분할 및 정복 (divide and conquer)
- 동적 프로그래밍 (dynamic programming)
- 힙우선순위큐 (Heap Priority Queue)
- 해시 (Hash), 해시테이블, 맵 (map) 자료구조
- 그래프 (Graph) 자료구조와 알고리즘

파이썬 기반 자료구조와 알고리즘

대표적인 자료구조와 특성

자료구조	특성
단순 배열	컴파일 단계에서 크기가 지정되어 변경되지 않는 크기의 배열.
동적 배열	프로그램 실행 단계에서 크기가 지정되며, 프로그램 실행 중에 크기를 변경할 수 있는 배열.
구조체 배열	구조체로 배열원소가 지정되는 배열.
연결형 리스트	확장성이 있으며, 단일 연결형 또는 이중 연결형으로 구성 가능. 다양한 컨테이너 자료구조의 내부적인 자료구조로도 활용됨.
스택	Last In First Out (LIFO) 특성을 가짐. 배열 또는 연결형 리스트로 구현할 수 있음.
큐	First In First Out (FIFO) 특성을 가짐. 배열 또는 연결형 리스트로 구현할 수 있음.
우선순위 큐	컨테이너 내부에 가장 우선순위가 높은 데이터 항목을 추출할 수 있도록 관리하며, 배열 또는 자기 참조 구조체로 구현할 수 있음.
이진 탐색 트리	컨테이너 내부에 포함된 데이터 항목들을 정렬된 상태로 관리하여야 할 때 매우 효율적임. 단순 이진 탐색 트리의 경우 편중될 수 있으며, 편중된 경우 검색 성능이 저하되기 때문에, 밸런싱이 필요함.
해시테이블 (Hash Table)	컨테이너 자료구조에 포함된 항목들을 문자열 (string) 또는 긴숫자를 키 (key)로 사용하여 관리하여야 하는 경우, key로부터 해시값을 구하고, 이 해시값을 배열의 인덱스로 사용함.
Map	key와 항목 간에 1:1 관계가 유지되어야 하는 경우에 사용되며, 해쉬 테이블을 기반으로 구현할 수 있음.
Dictionary	key와 항목 간에 1:N 관계가 유지되어야 하는 경우에 사용되며, 해쉬 테이블을 기반으로 구현할 수 있음.
trie	텍스트 검색을 신속하게 처리하며, 예측 구문 (predictive text) 제시, longest prefix matching 등에 활용됨.
그래프	정점 (vertex)/노드 (node)로 개체 (object)가 표현되고, 간선 (edge)/링크(link)들을 사용하여 개체 간의 관계를 표현하는 경우에 적합함. 그래프를 기반으로 경로 탐색, 최단 거리 경로 탐색, 신장트리 (spanning tree) 탐색 등에 활용됨.

파이썬 제공 기본 패키지

패키지/모듈	파이썬 제공 자료구조	관련 built-in 모듈, 기능 설명
기본 자료형	complex	복소수
	bytearry	바이트 배열
	list	리스트
	tuple	튜플
	dict	딕셔너리
	set	집합
array	array	기본 배열
NumPy	ndarray	다차원 배열
queue	queue	FIFO 큐
	lifo queue	stack과 같이 Last In First Out (LIFO) 기능 제공
	priority queue	(key, item)의 tuple을 저장하며, key 값이 작을수록 우선순위가 높은 순서를 유지
collections	namedtuple	키 (key)가 설정되는 튜플
	OrderedDict	정렬 기능이 포함되는 dict
	Counter	개수 파악하는데 특화된 클래스
	deque	double ended queue
sortedcontainer	SortedList	정렬이 유지되는 list
	SortedDict	정렬이 유지되는 dict
	SortedSet	정렬이 유지되는 집합 (set)

대표적인 알고리즘

알고리즘	주요 기능	전체 항목의 개수가 N인 경우의 알고리즘 복잡도
순차탐색	배열이나 연결형리스트에 포함되어 있는 항목을 차례대로 탐색	
이진탐색	정렬되어 있는 배열에 포함되어 있는 항목의 탐색 구간을 절반씩 줄여가며 탐색	
선택정렬	배열에 포함되어 있는 항목의 탐색 구간을 하나씩 줄여가며, 가장 작은(또는 큰) 항목을 찾아 그 구간의 맨 앞에 두는 동작을 반복하여 전체 항목들을 정렬.	
퀵정렬	배열에 포함되어 있는 항목의 탐색 구간을 중간 지점의 피봇 (pivot)을 중심으로 큰 항목들과 작은 항목들로 구분하여 정돈하며, (평균적으로) 절반씩 줄어든 탐색 구간에 대하여 재귀 함수 실행	
해시	문자열과 같이 길이가 일정하지 않는 키 (key)를 사용하여 해시값을 생성하고, 이를 배열의 인덱스나 데이터 항목이 저장되어 있는 그룹(버킷)을 찾는데 사용	해시코드 종류와 충돌 해결 방법에 따라 달라짐
그래프 깊이우선 탐색 (DFS)	그래프에서 지정된 시작 정점으로부터 목적지 정점까지의 경로를 깊이 우선 방식으로 탐색. 목적지까지의 탐색 시간은 상대적으로 빠를 수 있으나, 탐색된 경로가 최단 거리를 보장하지는 않음	
그래프 넓이우선 탐색 (BFS)	그래프에서 지정된 시작 정점으로부터 목적지 정점까지의 경로를 넓이 우선 방식으로 탐색. 탐색된 경로는 경유하는 간선수가 최소인 것을 보장함	
최단거리 경로 탐색	간선 (edge)에 가중치 (weight)이 지정된 그래프에서 시작 정점으로부터 목적지 정점까지의 경로 중에서 전체 거리가 가장 짧은 경로를 탐색. 탐색된 경로는 최단거리인 것을 보장함	
최소비용 신장트리 탐색	주어진 그래프에서 사용되는 간선들의 총 거리 (또는 가중치)가 가장 작으면서 모든 노드 (정점)들을 연결할 수 있는 최소비용 신장트리 (minimum spanning tree)를 탐색	

파이썬 제공 기본 알고리즘 (1)

패키지/모듈	파이썬 제공 알고리즘	관련 built-in 모듈, 기능 설명
re (regular expression)	search()	주어진 문자열에 포함된 패턴을 탐색
	match()	주어진 문자열로부터 지정된 패턴의 매칭을 확인
	fullmatch()	주어진 문자열로부터 지정된 패턴의 완전 매칭을 확인
	split()	주어진 문자열을 지정된 패턴으로 분할
	findall()	주어진 문자열에서 지정된 패턴을 모두 찾음
	sub()	주어진 문자열의 지정된 패턴을 지정된 대체 문자열로 치환
기본 내장 함수	sort()	List 정렬
heapq	heappush() heappop() heapreplace()	힙 큐 (우선순위 큐) 알고리즘
bisect	bisect(A, x)	시퀀스 자료형 (예: 리스트)에 대하여 이진 분할 알고리즘을 사용하며, 배열 A로부터 x의 위치를 찾아 반환
	insort(A, x)	리스트 A에 x를 순서에 맞추어 삽입
NumPy	ndarray.sort()	다차원 배열에 대한 정렬
	searchSorted()	정렬되어 있는 배열 A로부터 key 원소를 탐색하고, 그 위치를 반환

파이썬 제공 기본 알고리즘 (2)

패키지/모듈	파이썬 제공 알고리즘	관련 built-in 모듈, 기능 설명
NumPy Universal	amin()	주어진 배열에서 최솟값을 반환
	amax()	주어진 배열에서 최댓값을 반환
	fmax()	서로 다른 크기의 다차원 배열에서 큰 원소를 골라 배열을 구성
	fmin()	서로 다른 크기의 다차원 배열에서 작은 원소를 골라 배열을 구성
	where(condition[, x, y])	주어진 조건 (condition)에 따라 x 또는 y를 반환
	extract(condition, A)	주어진 조건을 만족하는 원소를 추출하여 반환
	shuffle()	주어진 배열의 원소 순서를 뒤섞어 줌
	sum()	주어진 배열의 합산 계산
	average()	주어진 배열의 평균 계산
	mean()	주어진 배열의 평균 계산
	var()	주어진 배열의 분산 (variance) 계산
	std()	주어진 배열의 표준 편차 (standard deviation) 계산
	cov()	주어진 배열의 공분산 (covariance)를 계산

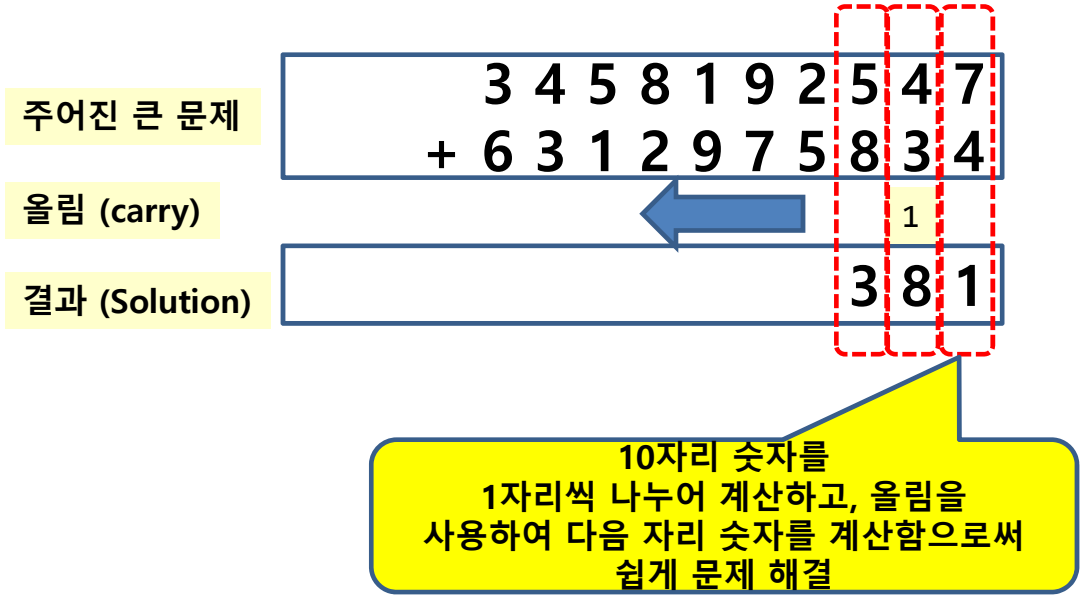
알고리즘의 성능 개선 (1)

- 분할 및 정복 (divide and conquer)

분할 및 정복 (Divide and Conquer)

◆ 분할 및 정복 알고리즘

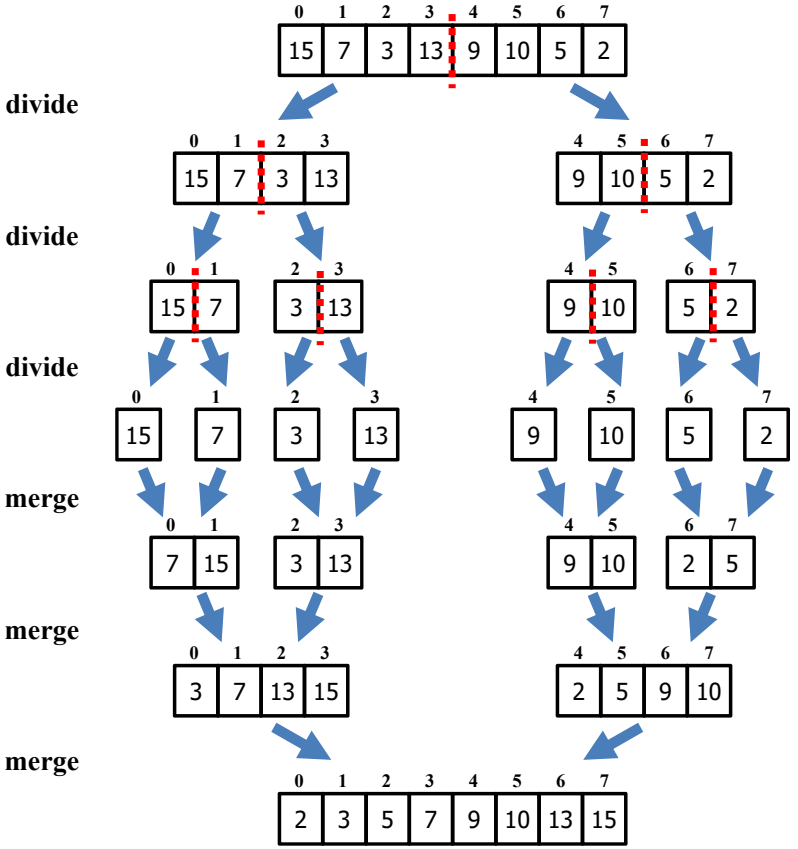
- 주어진 문제가 매우 큰 규모이기 때문에 한번에 다루기 힘든 경우 이 문제를 작은 단위로 나누어 분할하고 분할된 단위를 쉽게 해결하여 전체 해답을 찾는 방법
- 분할 및 정복 알고리즘의 예 – 10자리 정수의 덧셈



병합 정렬 (Merge Sort)

◆ 병합정렬 알고리즘

- 전체 알고리즘이 분할 (divide)과 병합(merge) 단계로 구성됨
- 분할 단계에서는 주어진 정렬 구간을 반복적으로 ½로 나누고, 최종적으로 2개씩의 원소가 남으면 이를 정렬
- 병합 단계에서는 정렬된 부분 구간들을 병합시키면서 정렬 기능을 수행
- 병합단계에서는 이미 부분 구간들이 정렬되어 있으므로 병합정렬이 신속하게 처리될 수 있음



Merge Sort

```
# mergeSort()
def merge(arr_left, arr_right):
    arr_result = []
    i, j = 0, 0
    while i < len(arr_left) and j < len(arr_right):
        if arr_left[i] < arr_right[j]:
            arr_result.append(arr_left[i])
            i += 1
        else:
            arr_result.append(arr_right[j])
            j += 1
    while (i < len(arr_left)):
        arr_result.append(arr_left[i])
        i += 1
    while (j < len(arr_right)):
        arr_result.append(arr_right[j])
        j += 1
    return arr_result

def mergeSort(arr): # merge_sort in increasing order
    if len(arr) < 2:
        return arr[:]
    else:
        middle = len(arr) // 2
        left = mergeSort(arr[:middle])
        right = mergeSort(arr[middle:])
        return merge(left, right)
```

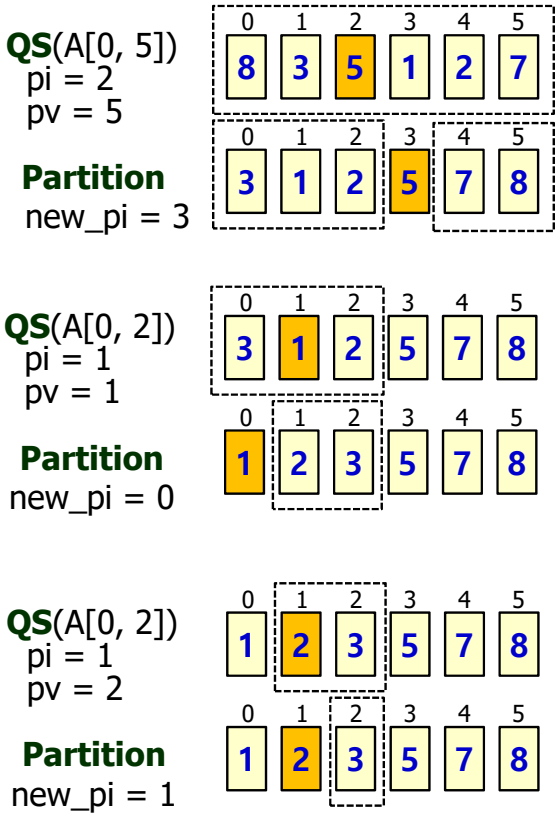
퀵정렬(Quick Sorting)

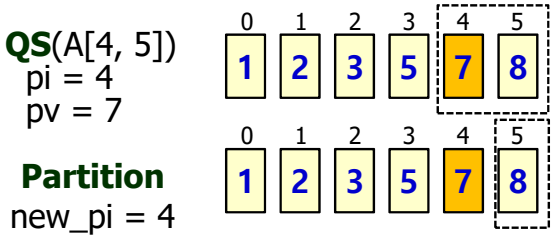
◆ 퀵 정렬 (Quick Sorting) 알고리즘

- 분할 및 정복 (divide and conquer) 방식의 알고리즘
- 탐색 구간의 중간 값을 pivot으로 선정하고, partition() 함수에서 이 pivot 보다 작은 원소들의 집합과 큰 원소들의 집합으로 분할 (pivot 원소의 위치는 변경될 수 있음)
- 분할된 각 구간에 대하여 quick_sort() 함수를 재귀함수 호출 (recursive function call)
- partition() 함수에서 pivot과의 비교 기능과 swapping 기능 수행
- partition() 함수를 사용한 분할 기능으로 탐색 구간을 ½ 정도씩으로 줄여감
- quick_sort() 함수의 재귀함수 호출에서 함수 호출의 오버헤드가 발생하며, 따라서 배열의 원소 개수가 작을 경우 선택정렬(selection sorting) 보다 늦은 성능을 가질 수 있음

퀵정렬 알고리즘의 수행 예

◆ Pivot Index, Pivot Vector와 Partition





Quick Sort

```
def _partition(arr, left, right, pi):
    pv = arr[pi]
    arr[pi], arr[right] = arr[right], arr[pi]
    new_pi = i = left
    while (i<=right-1):
        if (arr[i] <= pv):
            arr[new_pi], arr[i] = arr[i], arr[new_pi]
            new_pi += 1
        i += 1
    arr[new_pi], arr[right] = arr[right], arr[new_pi]
    return new_pi

def _quickSortLoop(arr, left, right):
    #print("quickSort", left, right)
    if (left >= right):
        return
    pi = (left + right)//2
    new_pi = _partition(arr, left, right, pi)
    #print("after partition : ", left, right, new_pi)
    if (left < new_pi -1):
        _quickSortLoop(arr, left, new_pi-1)
    if (new_pi + 1 < right):
        _quickSortLoop(arr, new_pi+1, right)

def quickSort(arr):
    size = len(arr)
    _quickSortLoop(arr, 0, size-1)
```

정렬 알고리즘의 성능 비교
- 선택정렬, 병합정렬, 퀵정렬

```
##################################
#  main() (1)
Selection_Sort_Limit = 50000

while True:
    A = array('i')
    print("\nComparisons of sorting algorithms")
    Arr_Size = int(input("Array Size (-1 to quit) = "))
    if Arr_Size == -1:
        break
    genRandArray(A, Arr_Size)

    # Testing Selection Sorting
    if Arr_Size <= Selection_Sort_Limit:

        np.random.shuffle(A)
        print("\nBefore Selection-Sort of A :")
        printArraySample(A, 10, 3)
        t1 = time.time()
        selectionSort(A, Arr_Size)
        t2 = time.time()
        print("After Selection-Sort of A .....")
        printArraySample(A, 10, 3)
        time_elapsed = t2 - t1
        print("Selection sorting took {} sec".format(time_elapsed))
```

```
#  main()  (2)
    # testing MergeSorting
    np.random.shuffle(A)
    print("\nBefore mergeSort of A :")
    printArraySample(A, 10, 3)
    t1 = time.time()
    A = mergeSort(A)
    t2 = time.time()
    print("After mergeSort of A :")
    printArraySample(A, 10, 3)
    time_elapsed = t2 - t1
    print("Merge sorting took {} sec".format(time_elapsed))

    # testing Quick Sorting
    np.random.shuffle(A)
    print("\nBefore quickSort of A :")
    printArraySample(A, 10, 3)
    t1 = time.time()
    quickSort(A)
    t2 = time.time()
    print("After quickSort of A :")
    printArraySample(A, 10, 3)
    time_elapsed = t2 - t1
    print("Quick sorting took {} sec".format(time_elapsed))
```

성능 측정 결과 (1)

```
Comparisons of sorting algorithms
Array Size (-1 to quit) = 50000

Before Selection-Sort of A :
  10367    2145    9892   29515   49772    3625   11005   14928    8276    8639
  32443    2997   37517   26958   23466   18683    4482   34025   48201   11739
  49121   16133   36520   47648   43711    4202   34088   42892   15377   22084
. . . . . .
  22749   30446   27583     536    6634   28596   30936   27038   27858   11709
  21731    8914   23713   30995   26440   23269    4691   10458   24916   23750
  29487   43421   21527   10747   31220   43996    5728    7894   41039   33582
After Selection-Sort of A .....
      0       1       2       3       4       5       6       7       8       9
     10      11      12      13      14      15      16      17      18      19
     20      21      22      23      24      25      26      27      28      29
. . . . . .
  49970   49971   49972   49973   49974   49975   49976   49977   49978   49979
  49980   49981   49982   49983   49984   49985   49986   49987   49988   49989
  49990   49991   49992   49993   49994   49995   49996   49997   49998   49999
Selection sorting took 130.39686012268066 sec

Before mergeSort of A :
  14153    1093    1444    3750    8921   48001   48003   21371   12764   17124
   3727   43248    1504   45610   22423   22756   14289   45657   28262   21605
  42646    7702    1019    1663    9334   10675   10744   21164   12326   24710
. . . . . .
  45298   44925   15079    6171   40494   46625   22246   31671   31781   33957
   6457    2322   39377   13208   44615   32954   13663   29201   39376   26428
  38267   14439   35329   43553   18882   28412   37447   35365   32038   12134

After mergeSort of A :
      0       1       2       3       4       5       6       7       8       9
     10      11      12      13      14      15      16      17      18      19
     20      21      22      23      24      25      26      27      28      29
. . . . . .
  49970   49971   49972   49973   49974   49975   49976   49977   49978   49979
  49980   49981   49982   49983   49984   49985   49986   49987   49988   49989
  49990   49991   49992   49993   49994   49995   49996   49997   49998   49999
Merge sorting took 0.2892262935638428 sec

Before quickSort of A :
  10406   22389   39387    8706   32181   27219   27901    7038   46701   39644
   1466   47241   15757   16231    2137   10341    1641   20592   27556   49587
  43783   41434   12694   36027   27402   33478   40904   41187   19375   10461
. . . . . .
   9035   17865   40672     876   27413   13135   36787    6922   44629   17968
  41898    2938   33939   16515   11505   19772   40243    3638   37302   19477
  24998   24086   12595    2527   35026   26734   30509   44103   31870   39152
After quickSort of A :
      0       1       2       3       4       5       6       7       8       9
     10      11      12      13      14      15      16      17      18      19
     20      21      22      23      24      25      26      27      28      29
. . . . . .
  49970   49971   49972   49973   49974   49975   49976   49977   49978   49979
  49980   49981   49982   49983   49984   49985   49986   49987   49988   49989
  49990   49991   49992   49993   49994   49995   49996   49997   49998   49999
Quick sorting took 0.22342252731323242 sec
```

성능 측정 결과 (2)

```
Comparisons of sorting algorithms
Array Size (-1 to quit) = 10000000
Before mergeSort of A :
8620717 2835834 2876780  100014 3255539 3598521 8507541 8359940 9079248  544792
8190097 8058223 9557997 2627929 8784117 6202907 3772059 5196081 7543830  409577
3628962 1797909 5072088  132968 5892326 3621279 3926664 3954695 5678731 3794646
. . . . . .
9929385 6354130 1964391 4719899 1528706 2605598 5036559 6899339 9346539  818064
7381665 8791561 8341871 7504605 5197217 6616676 2490091 5016436 7375580 4382561
5319106 9974321 3610393 6535665 4709921 3230667  349602 5688690 4377378 5693493
After mergeSort of A :
      0       1       2       3       4       5       6       7       8       9
     10      11      12      13      14      15      16      17      18      19
     20      21      22      23      24      25      26      27      28      29
. . . . . .
9999970 9999971 9999972 9999973 9999974 9999975 9999976 9999977 9999978 9999979
9999980 9999981 9999982 9999983 9999984 9999985 9999986 9999987 9999988 9999989
9999990 9999991 9999992 9999993 9999994 9999995 9999996 9999997 9999998 9999999
Merge sorting took 89.20360016822815 sec

Before quickSort of A :
 598604 4006011 3726601 2690827 3333810 2331063 7951612 5251719 5938350 7763875
2943865  794865 1013047 4795346 4318396 5306628 3175202 5696522 9597976 4345348
8511235 1601111 2882574 3269618 5371537 8149265 2426235 7777389 1706888 3349613
. . . . . .
8726035 7019870  577150 1907588 5290208 2673806 8099758 9344923 9404771 6417549
3895966 4569969 5331798 6652060 9966878 6727248 8243564  673396 4905916  335236
8912835 4102703 9578430 7505381 7670199 8605624 3351882 7968432 3834833 6000733
After quickSort of A :
      0       1       2       3       4       5       6       7       8       9
     10      11      12      13      14      15      16      17      18      19
     20      21      22      23      24      25      26      27      28      29
. . . . . .
9999970 9999971 9999972 9999973 9999974 9999975 9999976 9999977 9999978 9999979
9999980 9999981 9999982 9999983 9999984 9999985 9999986 9999987 9999988 9999989
9999990 9999991 9999992 9999993 9999994 9999995 9999996 9999997 9999998 9999999
Quick sorting took 85.12151026725769 sec
```

알고리즘의 성능 개선 (2)

- 동적 프로그래밍 (dynamic programming)

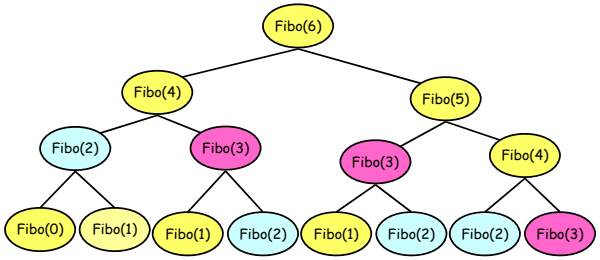
동적 프로그래밍 기법

◆ Dynamic Programming

- 동적 프로그래밍 (dynamic programming)은 동적 계획법 (dynamic planning)이라고도 하며, 다단계 의사 결정에서 각 단계에 따라 동적으로 변화하는 결정을 만드는 방법을 사용하도록 하는 것이다. 즉, 어떤 문제가 여러 단계로 반복되는 부분 문제로 이루어져 있을 때, 각 단계에 있는 부분 문제의 답을 기반으로 전체 문제의 답을 구하는 방법을 의미한다. 동적 계획법은 분할 및 정복 (divide and conquer)과는 다른 문제 해결 방법을 사용한다.
- **분할과 정복 기법**에서는 top-down 분할 방식으로 구성된 작은 단위의 문제는 개별적으로 하나의 문제로 다루어질 수 있는 구조를 가지도록 하는데, **동적 프로그래밍 기법**에서는 각 단계의 문제들은 그 이전 단계의 문제들의 해결 결과에 의존하도록 구성되며, 동일한 문제를 여러 번 다시 해결하느라 시간을 허비하지 않도록 한번 해결한 적이 있는 단계별 문제의 답을 표(table)에 저장하여 두고 이를 사용함으로써 전체 알고리즘 실행 시간을 단축시킨다.
- 동적 프로그래밍의 단계별 중간 답을 저장하는 표는 dict를 계산 중간결과를 저장하도록 구현할 수 있다.

동적 프로그램 구조의 재귀 피보나치 함수

```
# Recursive Fibonacci in dynamic programming structure (1)
import time

def srFibo(n): # simple recursive Fibonacci
    if n <= 1:
        return n
    else:
        return srFibo(n-1) + srFibo(n-2)

memo = dict()
def dynFibo(n): # dynamic programming Fibonacci
    if n in memo:
        return memo[n]
    elif n <= 1:
        memo[n] = n
        return n
    else:
        fibo_n = dynFibo(n-1) + dynFibo(n-2)
        memo[n] = fibo_n
        return fibo_n
```



memo = dict()

n	memo[n]
0	0
1	1
2	1
3	2
4	3
5	5
6	8
. . .	

동적 프로그램 구조의 재귀 피보나치 함수

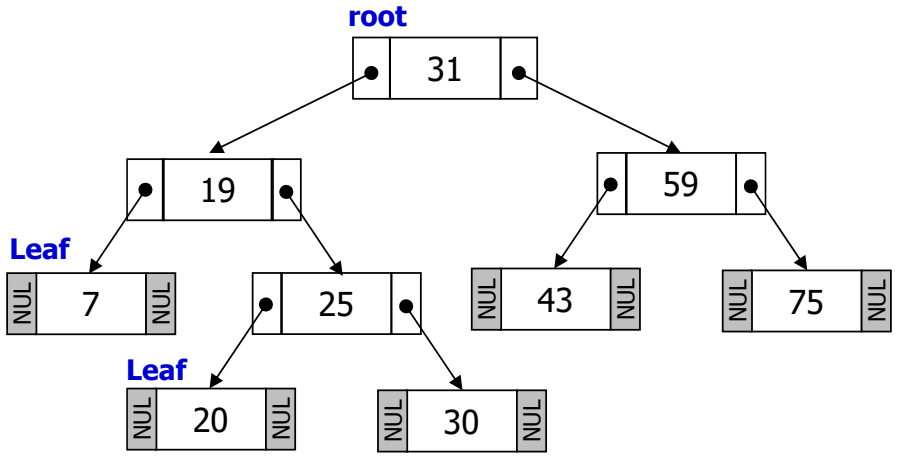
```
# Recursive Fibonacci in dynamic programming structure
#-----------------------------------
# Application
def main_Fibo(mode, start, end, step):
    for n in range(start, end, step):
        t_before = time.time()
        fibo_n = dynFibo(n)\
          if mode == "dyn" else srFibo(n)
        t_after = time.time()
        t_elapse = t_after - t_before
        t_elapse_ms = t_elapse * 1000
        print("Fibo({:3d}) = {:25d}, took {:10.2f} ms"\
              .format(n, fibo_n, t_elapse_ms))

if __name__ == '__main__':
 while True:
        print("Calculation of Fibonacci Series")
        mode = input("mode (dyn or sr, '.' to exit) = ")
        if mode == '.':
            break
        (start, end, step) = tuple(map(int, (input("start end step = ").split(' '))))
        main_Fibo(mode, start, end, step)
```

```
Calculation of Fibonacci Series
mode (dyn or sr, '.' to exit) = sr
start end step = 0 41 5
Fibo(  0) =                         0, took       0.00 ms
Fibo(  5) =                         5, took       0.00 ms
Fibo( 10) =                        55, took       0.00 ms
Fibo( 15) =                       610, took       0.00 ms
Fibo( 20) =                      6765, took       1.97 ms
Fibo( 25) =                     75025, took      26.92 ms
Fibo( 30) =                    832040, took     304.19 ms
Fibo( 35) =                   9227465, took    3353.03 ms
Fibo( 40) =                 102334155, took   37324.22 ms
Calculation of Fibonacci Series
mode (dyn or sr, '.' to exit) = dyn
start end step = 0 101 5
Fibo(  0) =                         0, took       0.00 ms
Fibo(  5) =                         5, took       0.00 ms
Fibo( 10) =                        55, took       0.00 ms
Fibo( 15) =                       610, took       0.00 ms
Fibo( 20) =                      6765, took       0.00 ms
Fibo( 25) =                     75025, took       0.00 ms
Fibo( 30) =                    832040, took       0.00 ms
Fibo( 35) =                   9227465, took       0.00 ms
Fibo( 40) =                 102334155, took       0.00 ms
Fibo( 45) =                1134903170, took       0.00 ms
Fibo( 50) =               12586269025, took       0.00 ms
Fibo( 55) =              139583862445, took       0.00 ms
Fibo( 60) =             1548008755920, took       1.00 ms
Fibo( 65) =            17167680177565, took       0.00 ms
Fibo( 70) =           190392490709135, took       0.00 ms
Fibo( 75) =          2111485077978050, took       0.00 ms
Fibo( 80) =         23416728348467685, took       0.00 ms
Fibo( 85) =        259695496911122585, took       0.00 ms
Fibo( 90) =       2880067194370816120, took       0.00 ms
Fibo( 95) =      31940434634990099905, took       0.00 ms
Fibo(100) =     354224848179261915075, took       0.00 ms
```

이진탐색트리 (Binary Search Tree)

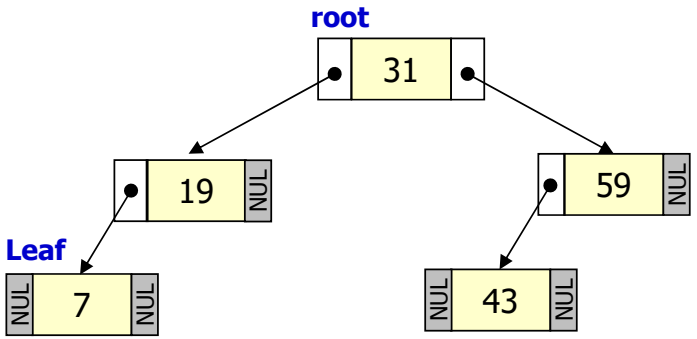
이진 탐색 트리 (Binary Search Tree)

- 이진 탐색 트리 (Binary search tree): 데이터 탐색을 쉽게 수행할 수 있도록 구성된 이진 트리
- 각 노드의 왼쪽 서브 트리는 그 노드의 값보다 작은 데이터를 가지는 노드들로만 구성
- 각 노드의 오른쪽 서브 트리는 그 노드의 데이터 값보다 같거나 큰 데이터 값을 가지는 노드들로만 구성



이진 탐색 트리 (Binary Search Tree)

- **Binary Search Tree (이진 탐색 트리)**
  - 비 선형 자료 구조이며, 각 노드는 0, 1, 또는 2개의 다른 노드를 가리킴
  - 부모와 자식 노드간에 크기 순서가 유지됨
    (왼쪽자식노드값 < 부모노드 값 ≤ 오른쪽 자식노드값)
- **Parent(부모) 노드, Children (자식) 노드**
  - (부모) 노드 N이 가리키는 하위 노드들은 그 (부모) 노드 N의 children (자식) 노드들임
- **Root(루트) 노드**
  - 이진 트리의 최상위 부모노드
- **Leaf(리프) 노드**
  - 이진 트리의 최하단 노드로서 자식이 없는 트리노드



이진 탐색 트리 (Binary Search Tree)의 연산

- 이진 탐색 트리의 생성 (create)
- 이진 트리에 노드를 삽입 (insert): 이진 트리에서의 데이터 값들의 순서가 유지되도록 올바른 곳에 새로운 노드를 삽입
- 이진 트리에서의 노드 탐색 (search): 지정된 데이터 값을 가지는 노드를 탐색
- 이진 트리로 부터의 노드 삭제 (delete, erase): 지정된 노드를 삭제하고, 나머지 노드 들을 순서에 맞추어 재배열

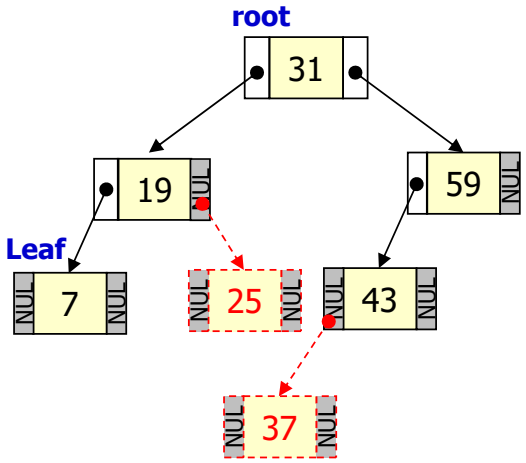
이진 탐색 트리 (Binary Search Tree)에서의 새로운 항목 삽입

◆ Binary Search Tree가 비어 있는 (empty) 경우

- 하나의 새로운 노드를 추가하여, **root** 노드로 만들고, 이 노드의 왼쪽 서브 트리와 오른쪽 서브 트리는 모두 비어 있는 상태로 구성

◆ Binary Search Tree가 비어 있지 않는 경우

- root 노드의 데이터 값과 새롭게 삽입되는 데이터를 비교함.
- 만약, 새로운 데이터가 root 노드 데이터 보다 작다면, 왼쪽 서브 트리에 추가하도록 하고,
- 새로운 데이터가 root 노드 데이터 보다 같거나 크다면, 오른쪽 서브 트리에 추가하도록 함.
- 각 서브 트리에서는 서브 트리 root와 비교하여 삽입될 위치를 결정하는 절차를 반복 수행하며, 정확한 위치를 결정한 후, 노드를 삽입함.



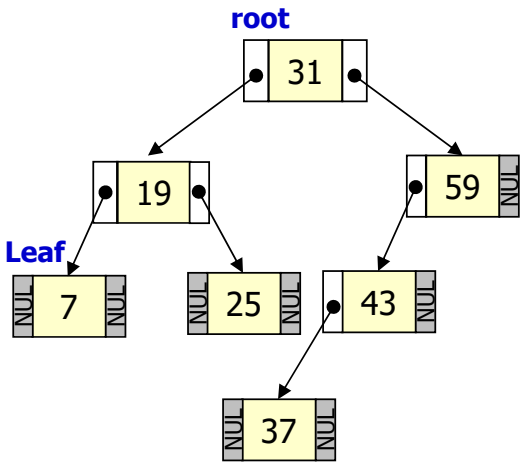
이진 탐색 트리에서의 탐색 (Searching)

1: Root 노드 부터 시작

2: 노드 데이터를 검사 :

- 2.1: 만약 찾고자 하는 데이터라면 탐색 성공/종료
- 2.2: 만약 탐색 데이터가 노드 데이터 보다 작다면, 왼쪽 서브 트리에서 step 2를 수행
- 2.3: 만약 탐색 데이터가 노드 데이터 보다 크다면, 오른쪽 서브 트리에서 step 2를 수행

3: 탐색 데이터를 찾을 때 까지 계속 탐색하거나, NULL 포인터에 도달할 때 까지 수행



Binary Search Tree

```
# Binary Search Tree (1)

class BinarySearchTree:
    class __Node:
        def __init__(self, val, left=None, right=None):
            self.val = val
            self.left = left
            self.right = right
        def getVal(self):
            return self.val
        def setVal(self, newVal):
            self.val = newVal
        def getLeft(self):
            return self.left
        def getRight(self):
            return self.right
        def setLeft(self, newLeft):
            self.left = newLeft
        def setRight(self, newRight):
            self.right = newRight
        def __iter__(self):
            if self.left != None:
                for elem in self.left:
                    yield elem
            yield self.val
            if self.right != None:
                for elem in self.right:
                    yield elem
    # end of class __Node
```

```
# Binary Search Tree (2)

# methods of class BinarySearchTree
    def __init__(self): # __init__() for class BinarySearchTree
        self._root = None
    def insert(self, val):
        def __insert(bst_node, val):
            if bst_node == None:
                return BinarySearchTree.__Node(val)
            if val < bst_node.getVal():
                bst_node.setLeft(__insert(bst_node.getLeft(), val))
            else:
                bst_node.setRight(__insert(bst_node.getRight(), val))
            return bst_node
        self._root = __insert(self._root, val)
    def __iter__(self):
        if self._root != None:
            return self._root.__iter__()
        else:
            return [].__iter__()
    def print_bst(self):
        def _print_bst(bst_node, level):
            if bst_node == None:
                return
            if bst_node.getRight() != None:
                _print_bst(bst_node.getRight(), level+1)
            for i in range(level):
                print("  ", end='')
            print(bst_node.getVal())
            if bst_node.getLeft() != None:
                _print_bst(bst_node.getLeft(), level+1)
        _print_bst(self._root, 0)
```

```
# Binary Search Tree (3)

def main():
    L = map(float, input("Enter a list of numbers : ").split())
    bst = BinarySearchTree()
    for x in L:
        bst.insert(x)
    for x in bst:
        print(x)
    print("\nBinary Search Tree with leveling:")
    bst.print_bst()

if __name__ == "__main__":
    main()
```

```
Enter a list of numbers : 5 3 7 6 2 1
1.0
2.0
3.0
5.0
6.0
7.0

Binary Search Tree with leveling:
 7.0
  6.0
5.0
 3.0
  2.0
   1.0
```

힙우선순위큐 (Heap Priority Queue)

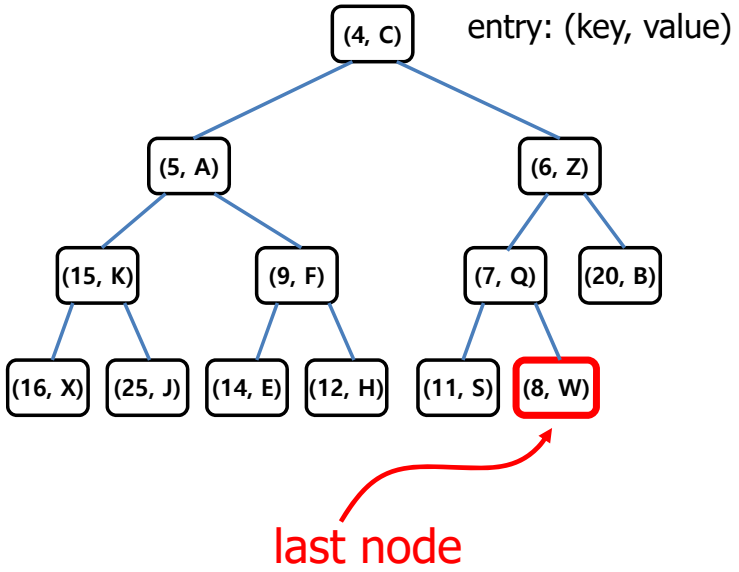
Priority Queue (우선순위큐)

- **우선순위큐 구성**
  - (키, 값) 튜플을 저장하며, 키는 우선순위를 나타냄
  - 키 값이 작을 수록 우선 순위는 높음
- **우선순위큐의 구현**
  - 완전이진트리 (complete binary tree) 기반의 힙우선순위큐 (heap priority queue)
  - 파이썬 queue 모듈의 queue.PriorityQueue()
- **우선순위큐 응용 분야:**
  - 주식거래 (stock market)
  - 경매 (auction)
  - 항공편의 대기 승객 우선 순위 관리

힙 (Heap)

- **힙(heap)**
  - 완전이진트리 (complete binary tree) 구조로 데이터를 저장
  - 힙순서(Heap-Order): 힙에 포함되는 모든 원소들의 순서

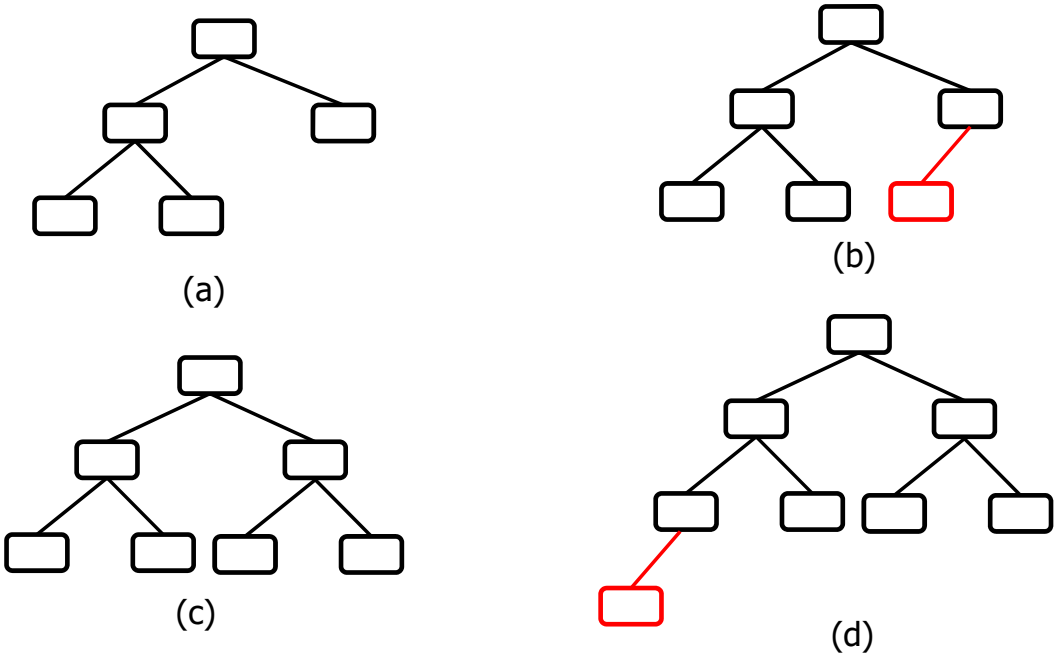
    $key(v) \geq key(parent(v))$
- **완전이진트리 (Complete Binary Tree):**
  - 힙의 높이 (height)가 h 일 때
  - depth $i = 0, \ldots, h - 1$에서 각각 최대 2^i 노드가 포함됨
  - depth $h - 1$에서 왼쪽에서 오른쪽으로 차례대로 채워짐
  - 힙의 마지막 노드 (last node)는 depth가 가장 깊은 (h-1)레벨의 가장 오른쪽 노드



완전이진트리 (Complete Binary Tree)

◆ 완전이진트리의 특성

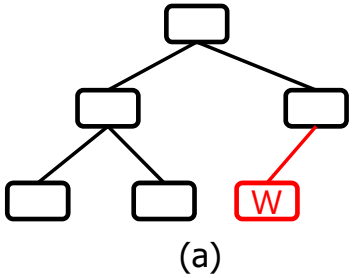
- 높이가 h인 힙 T는 완전이진트리로 구현되며, level = 0, 1, 2, ..., h-1
- 각 level (0 ≤ i ≤ h-1) 에서 최대 2^i 노드를 가지게 됨.
- level (h-1)에서 $2^{(h-1)}$ 개의 노드가 채워지면, level h가 구성되기 시작함



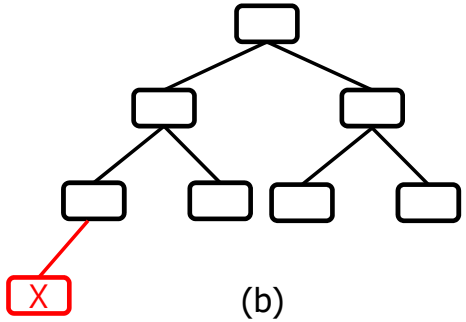
배열을 사용한 완전이진트리의 구현

◆ 배열 기반의 완전이진 트리 구현

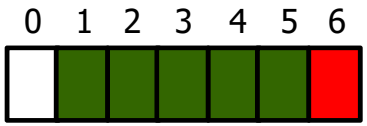
- 완전이진트리 (CBT)의 root가 v일 때, pos(v) = 1
- 만약 node u의 왼쪽 자식이 lc일 때, pos(lc) = 2 × pos(u)
- 만약 node u의 오른쪽 자식이 rc일 때, pos(rc) = 2 × pos(u) + 1



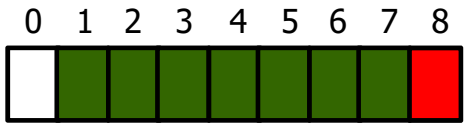
(a)



(b)



(c)

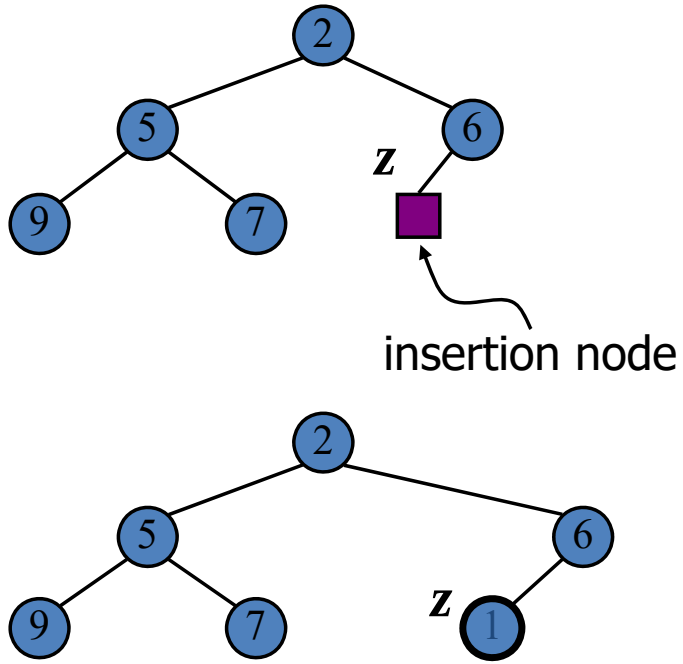


(d)

힙에 새로운 노드 삽입

◆ **힙에 새로운 key *k* 의 노드를 삽입할 때, 3 단계 실행**

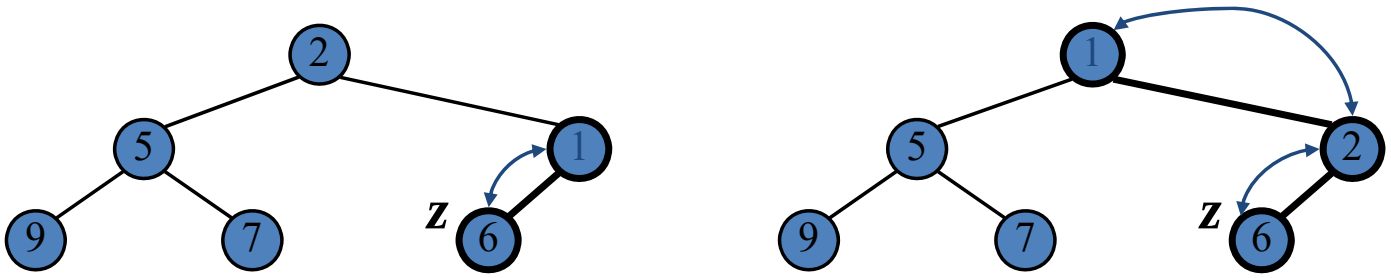
- 현재 힙 (완전이진트리)의 마지막 노드 (lost node)의 다음 위치 z 결정
- key K를 ***z*** 위치에 저장
- z 위치로 부터 root 노드까지의 경로에 있는 노드들을 heap order에 따라 정렬 (up heap bubbling)



Up-heap Bubbling

◆ Up-heap Bubbling

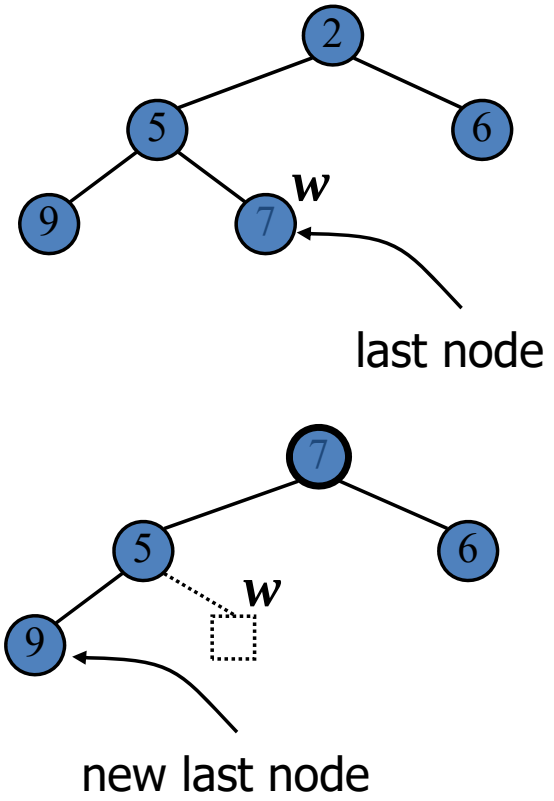
- 새로운 key k 를 노드 z에 삽입한 후, 노드 z로 부터 root 노드까지의 값들이 힙 정렬 순서 (heap order)가 아닐 수 있음
- Up-heap bubbling은 노드 z 에 저장된 key k 를 root노드까지의 경로상에 있는 다른 노드들과 비교하며, heap order가 유지될 수 있도록 조정
- Up-heap bubbling은 key k 가 root 노드에 도달하거나, 부모 노드의 key 값이 k보다 작을 때 중지됨
- 노드 개수가 n일 때, 힙은 $O(\log_2 n)$의 높이를 가지므로, up-heap bubbling은 최대 $O(\log_2 n)$ 번의 swapping이 실행될 수 있음



힙으로 부터 최우선 순위 key의 제거

◆ **힙에서 최우선 순위의 key를 제거할 때, 다음 3 단계로 실행**

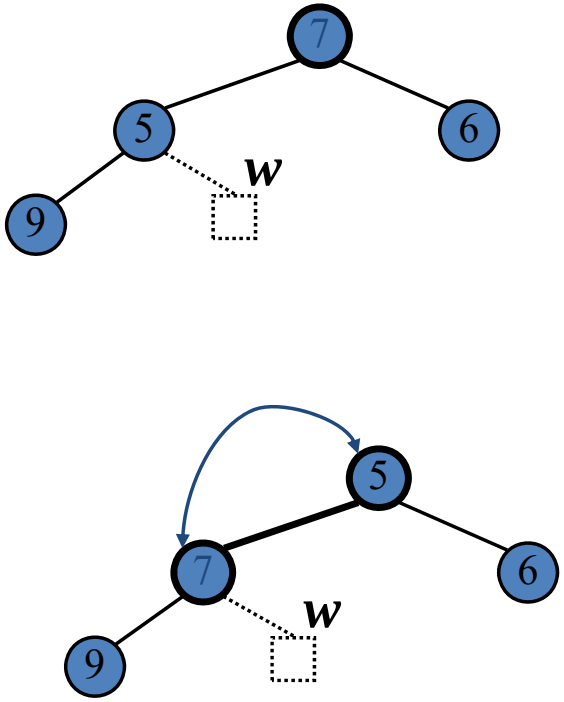
- 최우선 순위의 key를 root 노드로 부터 읽어 낸 후, last node 위치의 key ***w*** 를 root 노드 위치로 복사
- last node의 key ***w***를 삭제하고, 그 직전의 노드로 last node를 조정
- root 노드로 부터 하위 노드로 heap order로 재조정 (down-heap bubbling)



Down-heap Bubbling

◆ Down-heap Bubbling

- root 노드에 있는 최우선 순위 key *k* 를 제거한 후, last node의 key *w*를 root 노드 위치로 복사하면 root 노드로부터 하위 노드들 간에 heap order가 유지되지 않을 수 있음
- Down-heap bubbling은 root 노드와 하위 노드로의 힙 순서 (heap order)를 유지하도록 swapping을 실행하며 조정
- Down-heap bubbling은 key *k* 가 leaf 노드에 도달하거나 자식 노드가 key *k* 보다 더 큰 값일 경우 중지하게 됨
- 노드 개수가 n일 때, 힙은 $O(\log_2 n)$의 높이를 가지므로, down-heap bubbling은 최대 $O(\log_2 n)$ 번의 swapping이 실행될 수 있음



```
# Heap Priority Queue based on Complete Binary Tree (1)

class CompleteBinTree(object):
    def __init__(self):
        self._entry = [] # list
        self._root = 1
        self._end = 0
        self._entry.append(0) # entry[0] is not used in complete binary tree
    def __str__(self):
        #print("CBT::__str__() ....")
        if self._end == 0:
            s = "{} is empty now !!".format(self._name)
        else:
            s = ''
            for i in range(1, self._end+1):
                s += str(self._entry[i]) + ' '
        return s # insert new line


class HeapPriorityQueue(CompleteBinTree):
    def __init__(self, nm):
        CompleteBinTree.__init__(self)
        self._name = nm
    def __str__(self):
        s = CompleteBinTree.__str__(self)
        return s
```

```
# Heap Priority Queue based on Complete Binary Tree (2)

     def insert(self, item):
        self._end += 1
        self._entry.append(item)
        #print("HeapPrioQ : appending {} at index {}".format(item, self._end))

        idx = self._end
        # up-heap bubbling
        while idx != 1:
            #print("HeapPrioQ : Up-heap bubbling at idx({})".format(idx))
            par_idx = idx//2
            if (self._entry[par_idx] <= self._entry[idx]):
                break
            else:
                self._entry[par_idx], self._entry[idx] = self._entry[idx],
                  self._entry[par_idx]
            idx = par_idx
        return self._end

    def getHeapMin(self):
        return self._entry[self._root]
```

```
# Heap Priority Queue based on Complete Binary Tree (3)

    def removeMin(self):
        if self._end == 0:
            return None
        item_root = self._entry[1]
        self._entry[1] = self._entry[self._end]
        self._end -= 1

        # down-heap bubbling
        if self._end == 0: # now CBT became empty
            return item_root
        idx = 1
        while idx * 2 <= self._end:
            idx_c = idx * 2
            idx_rc = idx * 2 + 1
            if idx_rc <= self._end:
                if self._entry[idx_rc] < self._entry[idx_c]:
                    idx_c = idx_rc
            if self._entry[idx] <= self._entry[idx_c]:
                break
            else:
                self._entry[idx], self._entry[idx_c] = self._entry[idx_c],
                 self._entry[idx]
            idx = idx_c
        return item_root
```

```
# Heap Priority Queue based on Complete Binary Tree (3)

############################
# main()

hpriQ = HeapPriorityQueue("Heap Priority Queue")
print("hpriQ = ", hpriQ)
for item in range(5, 0, -1):
    print("hpriQ.insert({}) ...".format(item))
    hpriQ.insert(item)
    print("hpriQ = ", hpriQ)

for i in range(0, 5):
    item = hpriQ.removeMin()
    print("RemoveMin() of hpriQ = ", item)
    print("hpriQ = ", hpriQ)
```

```
hpriQ =  Heap Priority Queue is empty now !!
hpriQ.insert(5) ...
hpriQ =  5
hpriQ.insert(4) ...
hpriQ =  4 5
hpriQ.insert(3) ...
hpriQ =  3 5 4
hpriQ.insert(2) ...
hpriQ =  2 3 4 5
hpriQ.insert(1) ...
hpriQ =  1 2 4 5 3
RemoveMin() of hpriQ =  1
hpriQ =  2 3 4 5
RemoveMin() of hpriQ =  2
hpriQ =  3 5 4
RemoveMin() of hpriQ =  3
hpriQ =  4 5
RemoveMin() of hpriQ =  4
hpriQ =  5
RemoveMin() of hpriQ =  5
hpriQ =  Heap Priority Queue is empty now !!
```

queue.PriorityQueue()

```
# queue.PriorityQueue for event handling

import queue
import random
MAX_EVENTS = 100
priQ_event = queue.PriorityQueue(MAX_EVENTS)

def genEvents(num_events, priQ):
    for i in range(num_events):
        ev_title = "ev_" + str(i)
        ev_pri = random.randint(0, num_events)
        priQ_event.put((ev_pri, ev_title))
        print("Generated_event : (title({}), priority({}))"\
              .format(ev_title, ev_pri))

# ---------------
NUM_EVENTS = 10
genEvents(NUM_EVENTS, priQ_event)
print("after inserting {} events, priQ_event :"\
      .format(NUM_EVENTS))

for n in range(NUM_EVENTS):
    ev = priQ_event.get()
    print("priQ_event.get() : ", ev)
```

```
Generated_event : (title(ev_0), priority(7))
Generated_event : (title(ev_1), priority(4))
Generated_event : (title(ev_2), priority(4))
Generated_event : (title(ev_3), priority(4))
Generated_event : (title(ev_4), priority(10))
Generated_event : (title(ev_5), priority(9))
Generated_event : (title(ev_6), priority(8))
Generated_event : (title(ev_7), priority(3))
Generated_event : (title(ev_8), priority(10))
Generated_event : (title(ev_9), priority(9))
after inserting 10 events, priQ_event :
priQ_event.get() :  (3, 'ev_7')
priQ_event.get() :  (4, 'ev_1')
priQ_event.get() :  (4, 'ev_2')
priQ_event.get() :  (4, 'ev_3')
priQ_event.get() :  (7, 'ev_0')
priQ_event.get() :  (8, 'ev_6')
priQ_event.get() :  (9, 'ev_5')
priQ_event.get() :  (9, 'ev_9')
priQ_event.get() :  (10, 'ev_4')
priQ_event.get() :  (10, 'ev_8')
```

해시, 해시테이블, 맵 (map) 자료구조

해시 (hash)

◆ 해시 (hash)란?

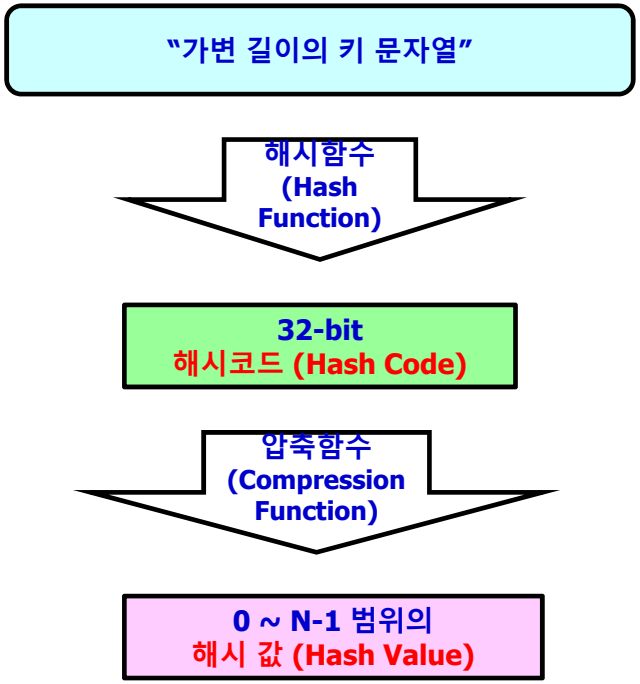
- **해시함수(hash function)**란 데이터의 효율적 관리를 목적으로 임의의 길이의 데이터를 고정된 길이의 데이터로 매핑하는 함수. 이 때 매핑 전 원래 데이터의 값을 키(key), 매핑 후 데이터의 값을 **해시값(hash value)**, 매핑하는 과정 자체를 **해싱(hashing)**라고 함.
- 해시 함수에 의해 얻어지는 값은 **해시 값**, **해시 코드**, **해시 체크섬** 또는 간단하게 **해시(hash)**라고 함
- 해시 함수의 용도 중 하나는 해시 테이블 (hash table) 자료구조에 사용되며, 매우 빠른 데이터 검색을 위한 컴퓨터 소프트웨어에 널리 사용됨
- 해시함수는 해쉬값의 개수보다 대개 많은 키값을 해쉬값으로 변환(many-to-one 대응)하기 때문에 해시함수가 서로 다른 두 개의 키에 대해 동일한 해시값을 내는 **충돌(collision)**이 발생할 수 있음

해시 값의 생성

- 일반적으로 해시값의 산출은 해시코드 계산 기능과 압축 함수를 차례로 사용함:
  - 해시 함수 (hash function):
    h_1: keys $\rightarrow$ integers
  - 압축 함수 (compression function):
    h_2: integers $\rightarrow$ [0, N - 1]

- 먼저 해시함수를 사용하여 주어진 키 (문자열)로부터 해시 코드를 생성하며, 다음으로 압축 함수를 사용하여 지정된 비트 수의 해시 값을 산출:
  $h(x) = h_2(h_1(x))$

- 생성된 해시 값은 주어진 비트 수의 범위에서 균등하게 분포하는 것이 좋은 성능 제공



Simple Hash Map

```
# simple hash map (1)
HashTableSize = 101
class SimpleHashMap:
    def __init__(self):
        self.hash_map = [None for h in range(HashTableSize)]
    def simple_hash(self, name):
        sum = 0
        for ch in name:
            sum += ord(ch)
            #print(name, sum, sum%HashTableSize)
        hash_value = sum % HashTableSize
        return hash_value
    def put(self, name, tel_no):
        self.hash_map[self.simple_hash(name)] = tel_no
    def print_hash_map(self):
        for idx, value in enumerate(self.hash_map):
            if value is not None:
                print("{:>3} {:5>}".format(idx, value))
    def search(self, name):
        value = self.hash_map[self.simple_hash(name)]
        return value
```

Simple Hash Map

```
# simple hash map (2)
#----------------------
HMap_telno = SimpleHashMap()
names = ["Kim Y-S", "Hong S-C", "Lee H-K", "Choi B-S", "ABCDEF", "CDEFAB"]
tel_numbers = [1234, 5678, 3456, 7890, 2468, 1357]
for i in range(len(names)):
    name = names[i]
    tel_no = tel_numbers[i]
    HMap_telno.put(name, tel_no)

HMap_telno.print_hash_map()

for idx in range(len(names)):
    name_to_search = names[idx]
    tel_no = HMap_telno.search(name_to_search)
    HMap_telno.search(name)
    print("tel_no of ({}) = {}".format(name_to_search, tel_no))
```

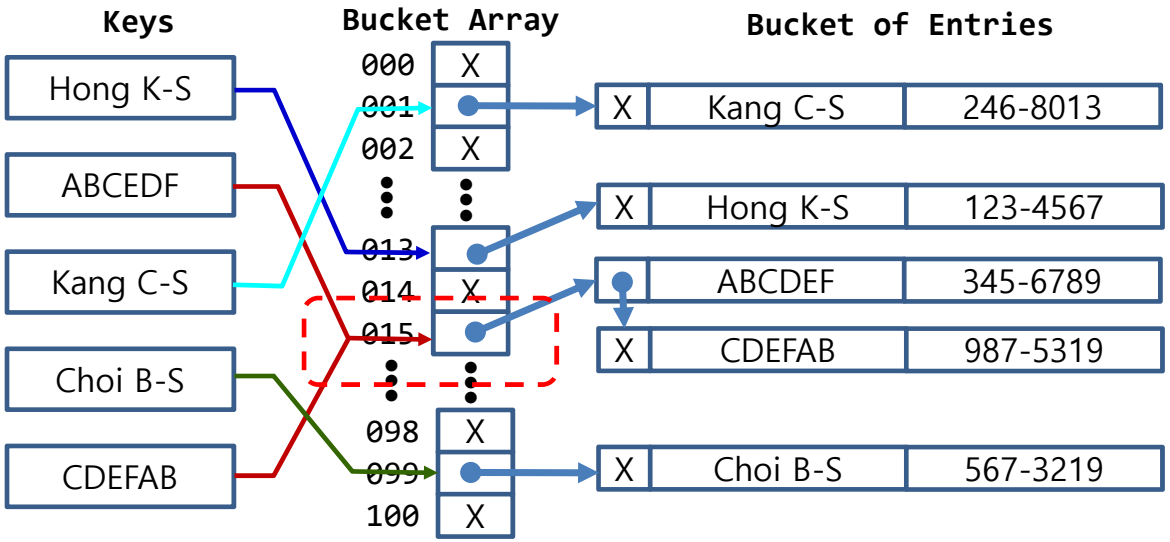
```
  1 1357
  7 7890
 17 5678
 33 1234
 98 3456
tel_no of (Kim Y-S) = 1234
tel_no of (Hong S-C) = 5678
tel_no of (Lee H-K) = 3456
tel_no of (Choi B-S) = 7890
tel_no of (ABCDEF) = 1357
tel_no of (CDEFAB) = 1357
```

Hash 충돌 발생

해시 충돌 해결

◆ 해시 충돌 및 해결 방법

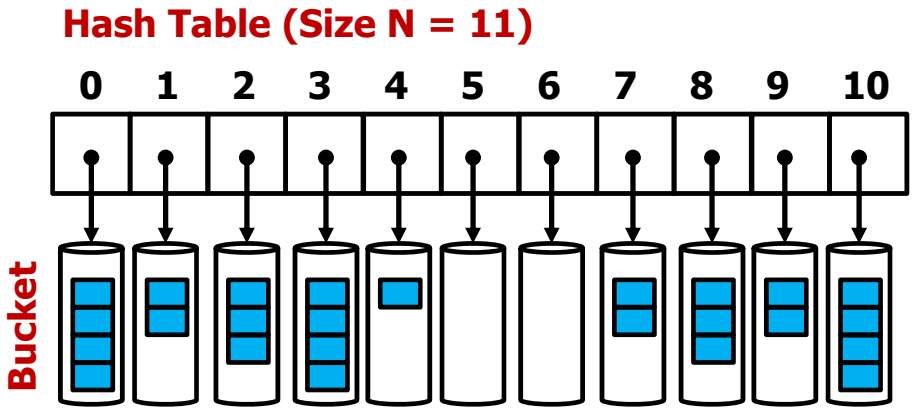
- 서로 다른 키 문자열로부터 생성된 해시 값이 동일한 경우 해시 충돌 발생
- 해결 방법 중 하나로 동일한 해시 값을 가진 모든 항목들을 연결형 리스트 (버킷)로 관리
- 해시 응용 분야에 따라 다양한 해결 방법이 있음



해시 충돌 해결을 위한 버킷

◆ 버킷을 사용하는 Hash Map

- Hash map에 버킷을 사용
- 각 버킷에는 동일한 해시 값을 가지는 항목들을 저장



HashMap_Bucket

```
# Hash Map with Bucket (1)

class Entry:
    def __init__(self, k, v):
        self._key = k
        self._value = v
    def __str__(self):
        return str(self._value)

def CyclicShiftHashCode(str_key): # str_key is string
    mask = (1 << 32) -1 # limit to 32-bit integers
    h = 0
    for ch in str_key:
        h = (h << 5 & mask) | (h >> 27) # cyclic shift hash code
        h += ord(ch)
    return h

class Bucket(Entry):
    def __init__(self):
        self._bucket = [] # implement bucket using list
    def _getitem(self, k):
        for item in self._bucket:
            if k == item._key:
                return item._value
        return None
    def _setitem(self, k, v):
        for item in self._bucket:
            if k == item._key:
                item._value = v
                return
        self._bucket.append(Entry(k, v))
```

HashMap_Bucket

```
# Hash Map with Bucket (2)

    def _delitem(self, k):
        for j in range(len(self._bucket)):
            if k == self._bucket[j]._key:
                self._bucket.pop(j)
                return 1
        return None
    def __len__(self):
        return len(self._bucket)
    def __iter__(self): # provides iterator as generator function
        for item in self._bucket:
            yield item._key

class HashMap_Bucket(Bucket):
    def __init__(self, table_size=11, prm=109345121):
        self._hash_tbl = table_size * [None]
        self._hash_tbl_size = table_size
        self._num_entry = 0
        self._prime = prm
    def _hash_value(self, k):
        return CyclicShiftHashCode(k) % self._prime % self._hash_tbl_size
    def __len__(self):
        return self._num_entry
    def _getitem(self, k):
        hv = self._hash_value(k)
        #print("key({}) => hash_tbl[{}]".format(k, hv))
        bucket = self._hash_tbl[hv]
        return bucket._getitem(k)
```

HashMap_Bucket

```
# Hash Map with Bucket (3)

    def _setitem(self, k, v):
        hv = self._hash_value(k)
        if self._hash_tbl[hv] is None:
            self._hash_tbl[hv] = Bucket()
        bucket = self._hash_tbl[hv]
        bucket._setitem(k, v)
    def _delitem(self, k):
        hv = self._hash_value(k)
        bucket = self._hash_tbl[hv]
        bucket._delitem(k)
        self._num_entry -= 1
    def __str__(self):
        s = ''
        for h in range(len(self._hash_tbl)):
            bucket = self._hash_tbl[h]
            if bucket is not None:
                s += "bucket[{:2d}] : ".format(h)
                for item in bucket:
                    s += str(item) + ", "
                s += "\n "
        return s

################################
#  main()
HashMap_Capacity = 7
print("Creating a HashMap_Bucket of capacity ({})".format(HashMap_Capacity))
hsMap = HashMap_Bucket(table_size = HashMap_Capacity)
```

HashMap_Bucket

```
# Hash Map with Bucket (4)

student_records = [("Kim Y-S", 2018, "ICE"), ("Park S-J", 2019, "CS"),\
                   ("Hong C-H", 2017, "EE"), ("Lee W-S", 2016, "ME"),\
                   ("Yoon C-M", 2015, "ICE"), ("Moon J-K", 2018, "CHEM")]
for i in range(len(student_records)):
    st_record = student_records[i]
    st_name = st_record[0]
    hsMap._setitem(st_name, st_record)
print("Current HashMap Internal Structure:\n", hsMap)

print("Checking entry searching in HashMap")
for i in range(len(student_records)):
    st_name = student_records[i][0]
    st_record = hsMap._getitem(st_name)
    print("key ({}) : Value ({})".format(st_name, st_record))
```

```
Creating a HashMap_Bucket of capacity (7)
Current HashMap Internal Structure:
 bucket[ 2] : Kim Y-S, Yoon C-M, Moon J-K,
 bucket[ 3] : Park S-J, Hong C-H,
 bucket[ 4] : Lee W-S,

Checking entry searching in HashMap
key (Kim Y-S) : Value (('Kim Y-S', 2018, 'ICE'))
key (Park S-J) : Value (('Park S-J', 2019, 'CS'))
key (Hong C-H) : Value (('Hong C-H', 2017, 'EE'))
key (Lee W-S) : Value (('Lee W-S', 2016, 'ME'))
key (Yoon C-M) : Value (('Yoon C-M', 2015, 'ICE'))
key (Moon J-K) : Value (('Moon J-K', 2018, 'CHEM'))
```

그래프 자료구조와 알고리즘

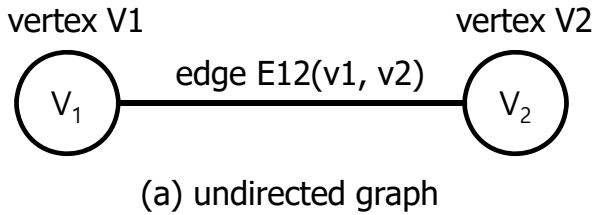
Vertex/Node, Edge

◆ 정점 (Vertex), 노드(Node)

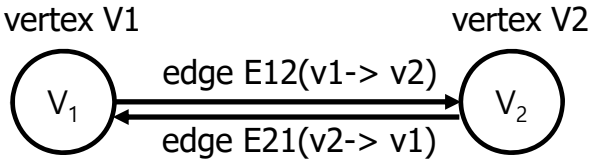
- 지도상의 도시 (city), 공항
- 연관성 분석 대상의 정보

◆ 간선 (Edge)

- 도시를 연결하는 도로, 항공편
- 분석 대상 정보간의 연관성
- 방향성 간선 또는 무방향성 간선으로 표현

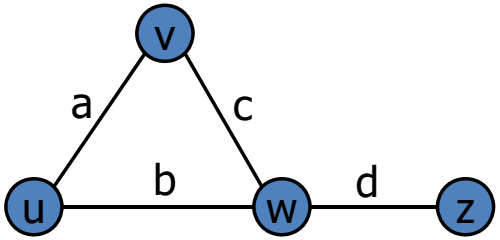


(a) undirected graph



(b) directed graph

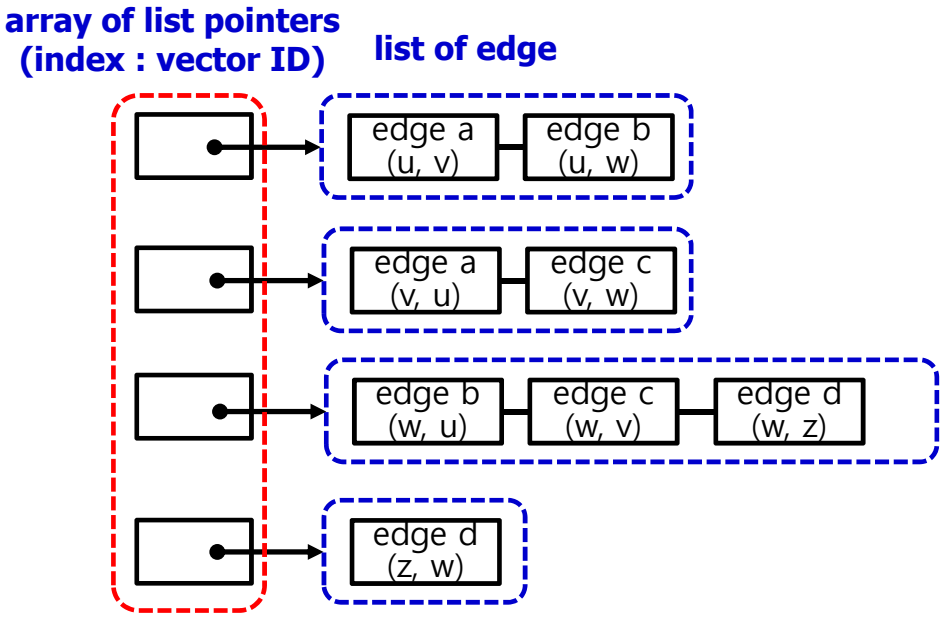
그래프의 연결성 표현



(a) 그래프 토폴로지 **(Graph Topology)**

Vertex (u, 0)
Vertex (v, 1)
Vertex (w, 2)
Vertex (z, 3)

(b) 정점 배열 **(Vertex Array)**



(c) 인접리스트 배열 **(Adjacency List Array)**

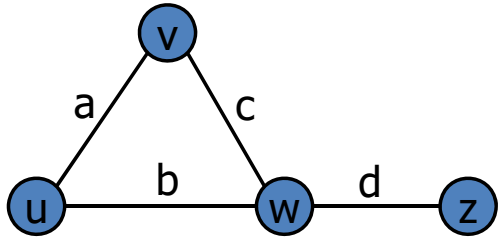
인접행렬 (Adjacency Matrix), 거리표

◆ 2차원 인접 행렬 (2-D adjacency matrix)

- 각 노드간의 간선 정보를 표로 정리

◆ 거리표 (Distance Table)

- 만약 두 노드를 직접 연결하는 간선이 없는 경우: ∞
- 출발지와 도착지가 동일 노드인 경우: 0
- 두 노드간에 간선이 있는 경우 : 간선의 가중치



	u	v	w	z
u	0	a	b	∞
v	a	0	c	∞
w	b	c	0	d
z	∞	∞	d	0

그래프 탐색 (Graph Search)

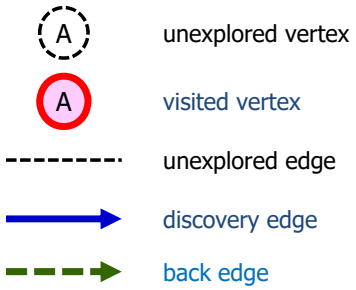
◆ 깊이 우선 탐색 (Depth First Search)

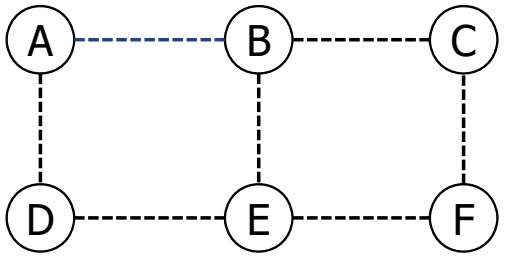
- 시작 노드 (start node)로 부터 간선이 연결되어 있는 새로운 노드를 찾아 깊이 우선으로 탐색
- 목적지까지의 경로가 존재하는지를 파악하는 것에 중점
- 탐색된 결과의 경로가 최단 거리 경로를 보장하지 않음

◆ 넓이 우선 탐색 (Breadth First Search)

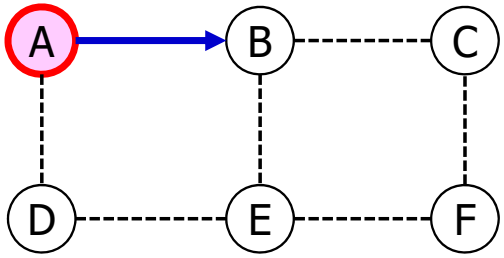
- 현재 방문중인 노드에 연결되어 있는 모든 노드들을 확인한 후, 다음 레벨의 노드들을 탐색
- 레벨 0: 시작노드
- 레벨 1: 시작노드로 부터 하나의 간선을 통하여 이동할 수 있는 노드 그룹
- 레벨 2: 시작노드로 부터 두 개의 간선을 통하여 이동할 수 있는 노드 그룹
- 넓이우선탐색으로 탐색된 경로는 간선의 개수가 최소가 되는 경로

그래프의 깊이우선탐색 (Depth First Search) (1)

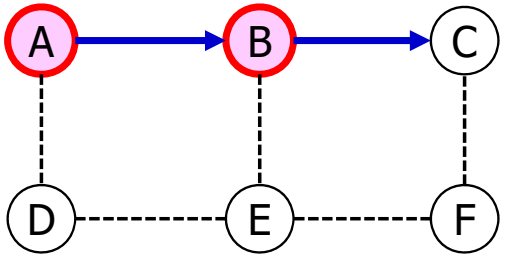




(a) Graph to be searched

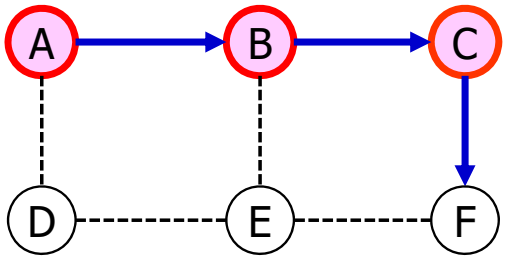


(b) Vertex A selected,
Edge (A-B) searched

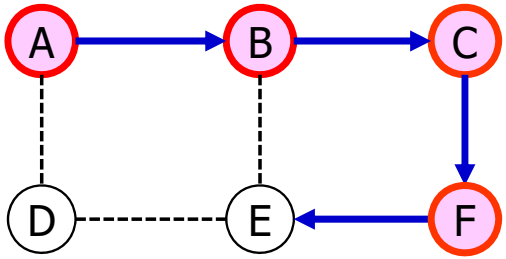


(c) Vertex B selected,
Edge (B-C) searched

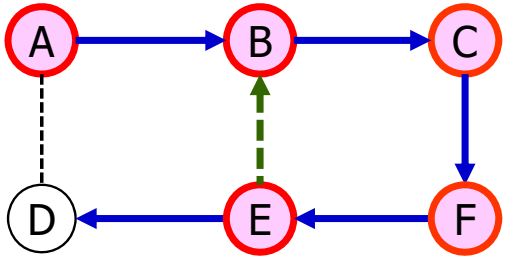
그래프의 깊이우선탐색 (Depth First Search) (2)



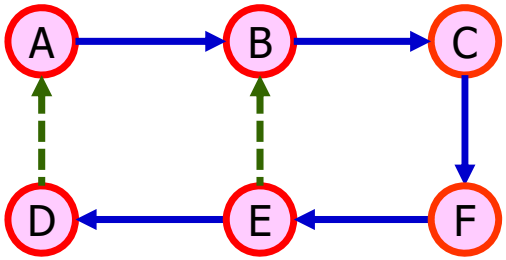
(d) Vertex C selected,
Edge (C-F) searched



(e) Vertex F selected,
Edge (F-E) searched

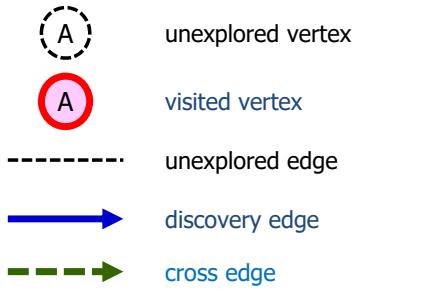


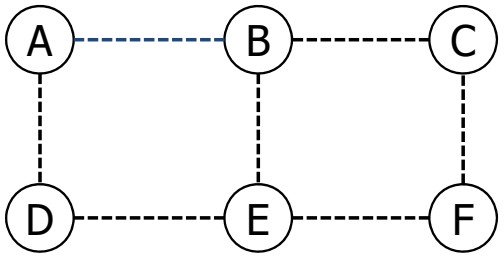
(f) Vertex E selected,
Edge (B-E) and Edge (E-D) searched



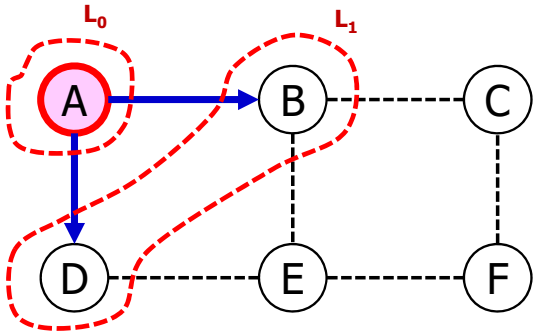
(g) Vertex D selected,
Edge (D-A) searched

그래프의 넓이우선탐색 (Breadth First Search) (1)

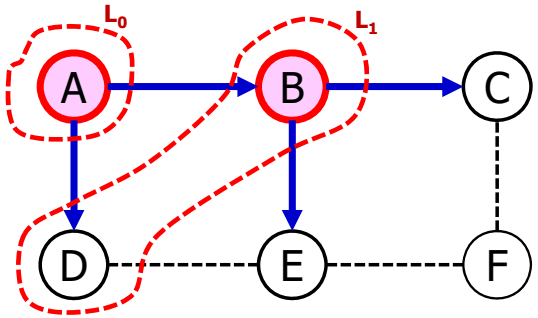




(a) Graph to be searched

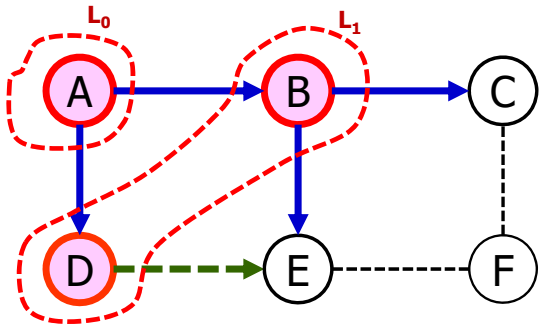


(b) Vertex A selected,
Edges (A-B), (A_D) searched

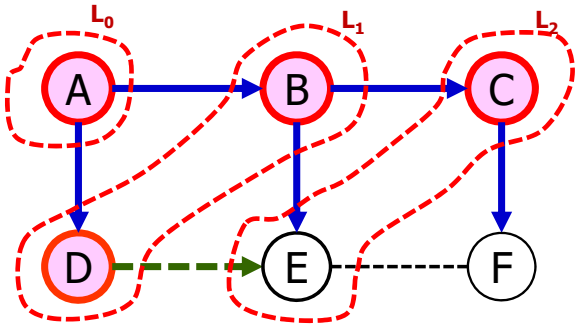


(c) Vertex B selected,
Edges (B-C), (B-E) searched

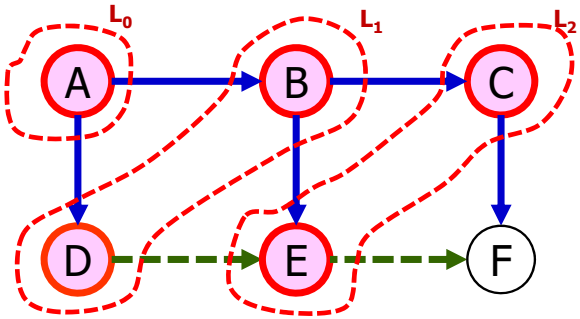
그래프의 넓이우선탐색(Breadth First Search) (2)



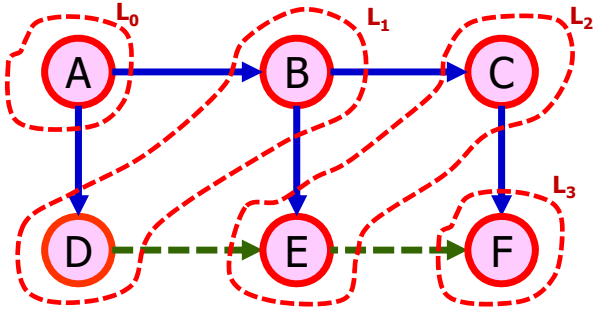
(d) Vertex D selected,
Edges (D-E) searched



(e) Vertex C selected,
Edges (C-F) searched



(f) Vertex E selected,
Edges (B-C), (B-E), (D-E) searched



(g) Vertex F selected

가중치 그래프 (weighted graph)에서 최단거리 경로찾기

◆ 가중치 그래프 (weighted graph)

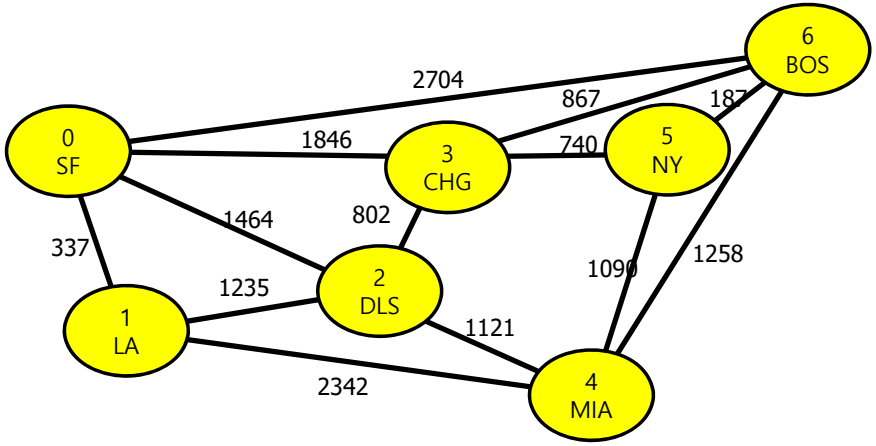
- 그래프의 간선에 가중치 (거리, 시간, 비용)가 설정됨

◆ 최단거리 경로 탐색 (shorted distance path)

- 가중치 그래프 상에서 지정된 두 정점간의 경로 중 최단 거리 경로의 누적된 거리가 최소가 되는 경로

◆ 주요 응용 분야

- 자동차 네비게이션에서 최단 거리 경로 탐색
- 인터넷에서의 경로 설정
- 항공 여행에서의 항공편 설정



다익스트라 알고리즘 (Dijkstra's Algorithm)

◆ Dijkstra's Algorithm

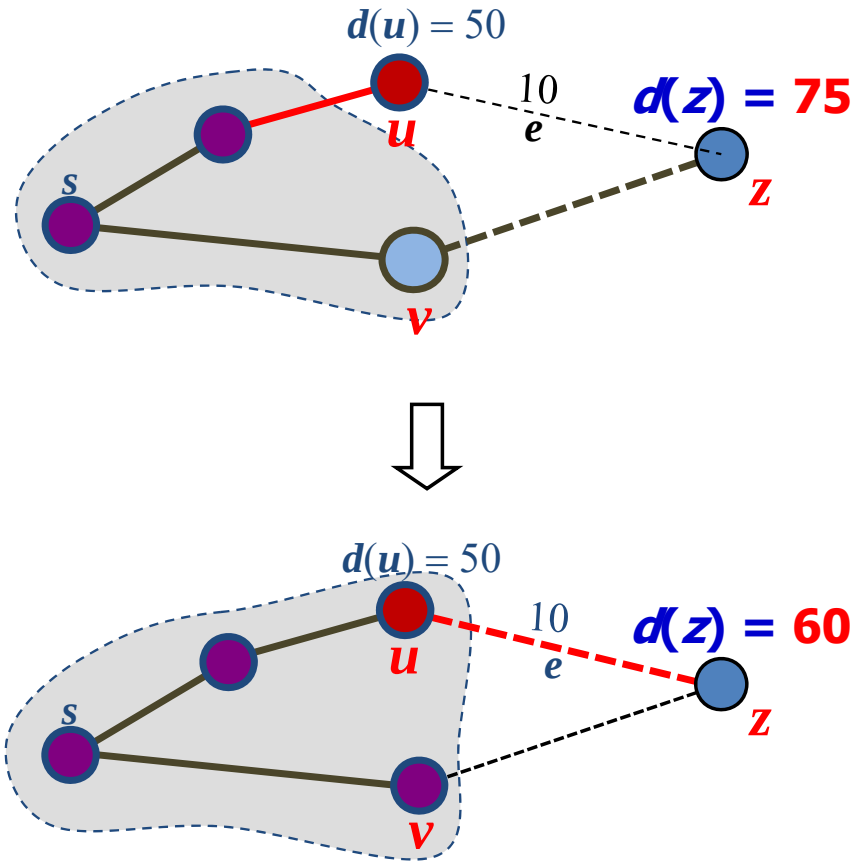
- 그래프의 간선들에 가중치 (weight)가 설정된 그래프에서 최단거리 경로 탐색

Algorithm Dijkstra_Shortest_Path(G, start, target)
전달인수: 가중치가 할당된 그래프 G, 시작노드 start, 목적지노드 target
반환값: 시작노드에서 목적지 노드까지의 최단 거리 경로
1: 시작노드를 기준으로 거리표 (distance table)의 초기화
2: 모든 노드를 "Not_Selected"상태로 설정
3: 시작노드를 "Selected(선택)"로 설정
4: 시작노드로부터 "Not_Selected(비선택)" 노드까지의 누적거리를 계산
5: while (선택되지 않은 노드가 없을 때까지):
6: 현재 "Not_Selected" 노드 중 누적거리가 가장 짧은 노드를 선택하여 "Selected"로 설정
7: 만약 새롭게 선택된 노드가 목적지 노드이면 현재까지의 경로와 누적 거리를 반환하여 알고리즘 종료
8: 새롭게 선택된 노드를 경유하여 도달할 수 있는 노드의 누적 거리를 계산하여, 기존 경로보다 누적 비용이 더 작은 경우 경로 갱신 (Edge relaxation)

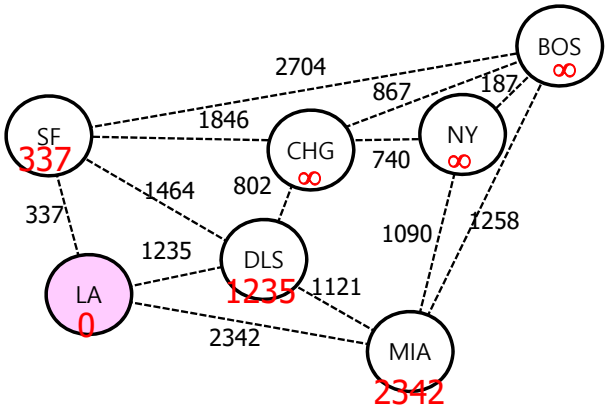
Edge Relaxation

- **Dijstra 알고리즘에서 새롭게 선택된 노드 u의 간선 *e* = (*u*, *z*)에 연결된 노드 z**
  - *u* 는 새롭게 선택된 노드
  - *z* 는 아직 선택되지 않은 노드
- **간선 e를 통한 노드 z의 누적 거리가 더 좋은 조건인 경우, 경로 갱신 :**

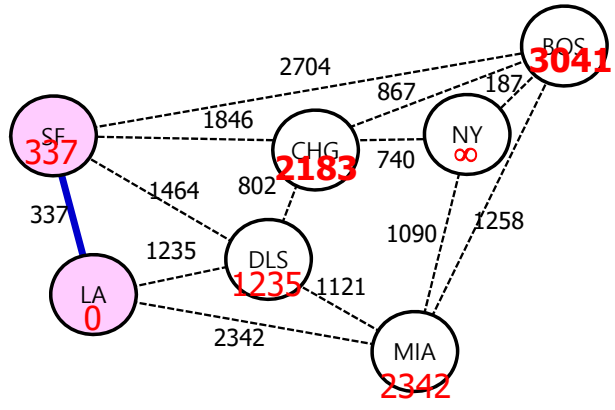
$$d(z) \leftarrow \min\{ d(z),\ d(u) + weight(e) \}$$



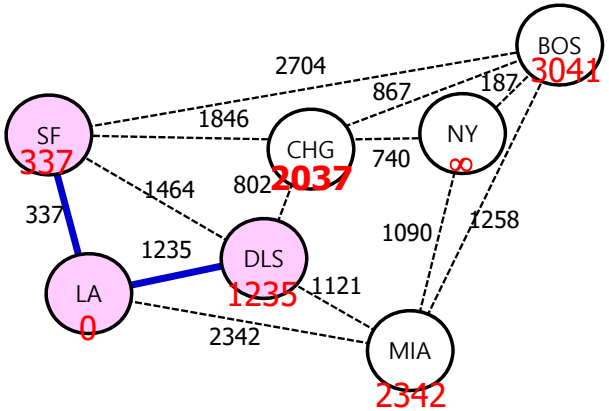
가중치가 설정된 그래프에서의 Dijkstra 알고리즘 실행 예 (1)



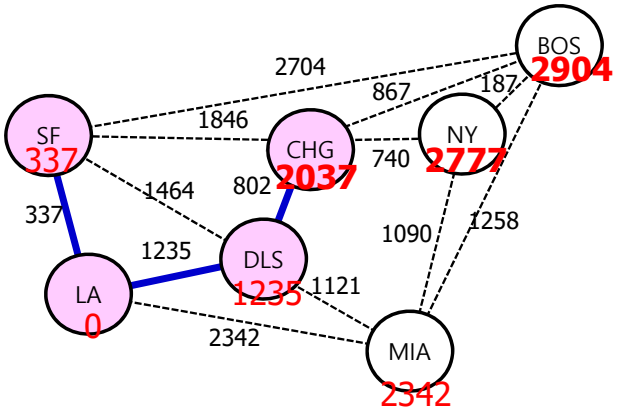
(a) round 0



(b) round 1

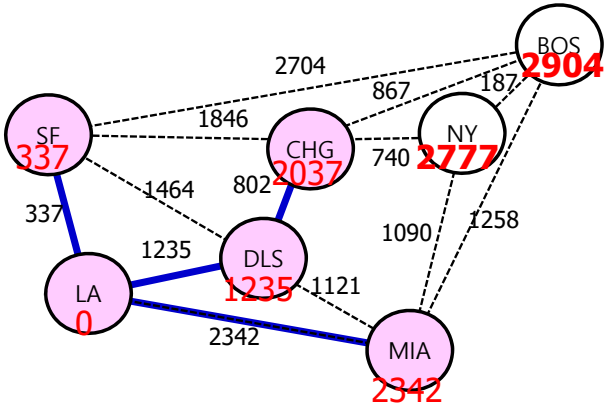


(c) round 2

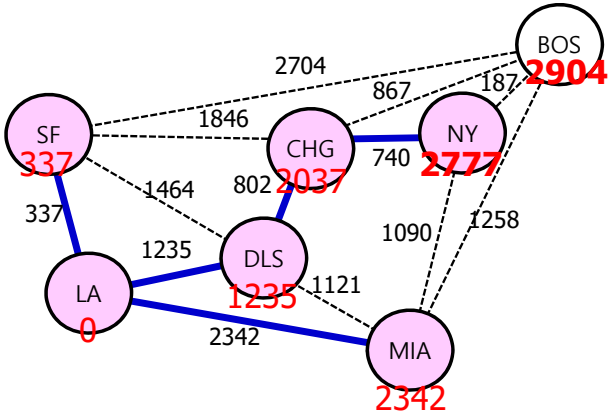


(d) round 3

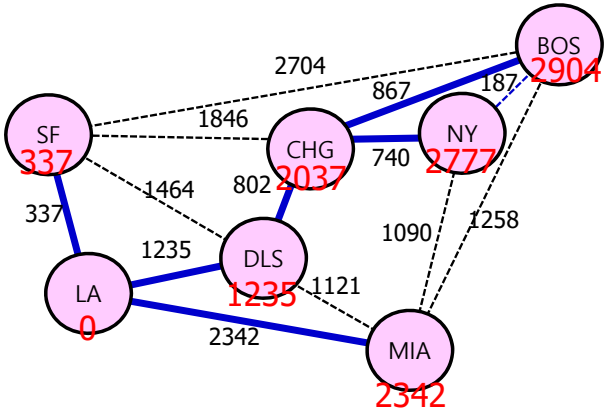
가중치가 설정된 그래프에서의 Dijkstra 알고리즘 실행 예 (2)



(e) round 4



(f) round 5



(g) round 6

그래프 자료구조와 알고리즘의 구현

그래프 알고리즘 구현

```
# Graph with DFS, BFS, Dijkstra, MST(MinimumSpanningTree) (1)
import sys # for sys.maxsize as PLUS_INF

class Node(object):
    def __init__(self, name):
        self.name = name
    def getName(self):
        return self.name
    def __str__(self):
        return self.name

class Edge(object):
    def __init__(self, src_nm, dest_nm):
        self.src_nm = src_nm
        self.dest_nm = dest_nm
    def getSource_nm(self):
        return self.src_nm
    def getDestination_nm(self):
        return self.dest_nm
    def __str__(self):
        return "{:3}->{:3}".format(self.src_nm, self.dest_nm)

class WeightedEdge(Edge):
    def __init__(self, src_nm, dest_nm, weight=1.0):
        Edge.__init__(self, src_nm, dest_nm)
        self.weight = weight
    def getWeight(self):
        return self.weight
    def __str__(self):
        return "{:3}->({:3})->{}".format(self.src_nm, self.weight, self.dest_nm)
```

```
# Graph with DFS, BFS, Dijkstra, MST(MinimumSpanningTree) (2)

class WeightedGraph(object):
    def __init__(self):
        self.nodes = [] # list of Node(v_id, v_names)
        self.node_names = []
        self.wedges = []  # list of weighted_edges
        self.adjacencyList = {} # dict of {src_nm:list of node_names)
        self.edgeWeights = {} # dictionary of {edge(src_nm, dest_nm):weight}
    def addNode(self, node):
        if node in self.nodes:
            raise ValueError("Duplicated node")
        else:
            self.nodes.append(node)
            node_nm = node.getName()
            self.node_names.append(node_nm)
            self.adjacencyList[node_nm] = []
    def addEdge(self, weighted_edge):
        src_nm = weighted_edge.getSource_nm()
        dest_nm = weighted_edge.getDestination_nm()
        if not (src_nm in self.node_names and dest_nm in self.node_names):
            raise ValueError("Node not in graph")
        self.wedges.append(weighted_edge)
        self.adjacencyList[src_nm].append(dest_nm)
        self.edgeWeights[(src_nm, dest_nm)] = weighted_edge.getWeight()
    def getNeighbors(self, node_nm):
        #print(" WeightedGraph::getNeighbors({}) = {}".format(node_nm, self.adjacencyList[node_nm]))
        return self.adjacencyList[node_nm]
    def getAdjacencyList(self):
        return self.adjacencyList
    def getNode_NMs(self):
        return self.node_names
    def getWEdges(self):
        return self.wedges
```

```
# Graph with DFS, BFS, Dijkstra, MST(MinimumSpanningTree) (3)

    def getEdgeWeight(self, edge):
        if (edge.src_nm, edge.dest_nm) in self.edgeWeights:
            return self.edgeWeights[(edge.src_nm, edge.dest_nm)]
        else:
            None
    def printConnectivity(self):
        for node_nm in self.node_names:
            print(" AdjacencyList[{}] = {}"\
               .format(node_nm, self.adjacencyList[node_nm]))
    def printEdges(G):
        eCount = 0
        for e in edges:
            print("  {}".format(e), end=', ')
            eCount += 1
            if eCount % 5 == 0:
                print()
    def __str__(self):
        result = ''
        for src in self.nodes:
            for dest in self.edges[src]:
                result = result + src.getName() + '->'\
                         + dest.getName() + '\n'
                return result[:-1] # omit final newline

def printPath(path):
    result = ''
    for i in range(len(path)):
        result += str(path[i])
        if i != len(path) -1:
            result += '->'
    return result
```

```
# Graph with DFS, BFS, Dijkstra, MST(MinimumSpanningTree) (4)

def DFS(graph, begin_nm, end_nm, path, shortest): # depth first search
    #print("DFS:: begin({}), end({}))".format(begin_nm, end_nm))
    path.append(begin_nm)
    #print("Current DFS path : ", printPath(path))
    if begin_nm == end_nm:
        return path
    for node_nm in graph.getNeighbors(begin_nm):
        if node_nm not in path: # avoid cycle
            if shortest == None or len(path) < len(shortest):
                newPath = DFS(graph, node_nm, end_nm, path, shortest)
                if newPath != None:
                    shortest = newPath
    return shortest
```

```
# Graph with DFS, BFS, Dijkstra, MST(MinimumSpanningTree) (5)

def BFS(graph, start_nm, end_nm): # breadth first search
    initPath = [start_nm]
    pathQueue = [initPath]
    #count = 0
    while len(pathQueue) != 0:
        #print("Round ({:2d}) - pathQueue : ".format(count), end=' ')
        """
        for path in pathQueue:
            print("{}".format(printPath(path)), end=' ')
        print()
        """
        tmpPath = pathQueue.pop(0)
        #print(" - current tmpPath: {}".format(printPath(tmpPath)))
        lastNode_nm = tmpPath[-1]
        if lastNode_nm == end_nm:
            return tmpPath
        for nextNode_nm in graph.getNeighbors(lastNode_nm):
            if nextNode_nm not in tmpPath:
                newPath = tmpPath + [nextNode_nm]
                pathQueue.append(newPath)
        #count += 1
    return None
```

```
# Graph with DFS, BFS, Dijkstra, MST(MinimumSpanningTree) (5)

PLUS_INF = sys.maxsize # define as max of integer
def Dijkstra(G, start_nm, end_nm): # Dijkstra shortest path
    errorInLoop = False
    nodeAccWeight= {} # dictionary of node:accumulated_weight_from_start
    nodeStatus = {}
    prevNodes_nm = {} # previous node in the path from the start to the end
    selectedNodes = []
    remainingNodes = []
    wEdges = G.getWEdges()
    #print("Dijkstra::edges : ", edges)
    for node_nm in G.node_names:
        e = Edge(start_nm, node_nm)
        if node_nm == start_nm:
            eWeight = 0
        else:
            eWeight = G.getEdgeWeight(e)
            if eWeight == None:
                eWeight = PLUS_INF
        print(" Initial weight of edge ({}) = {}: ".format(e, eWeight))
        nodeAccWeight[node_nm] = eWeight
        nodeStatus[node_nm] = False # not selected yet
        prevNodes_nm[node_nm] = start_nm
        if node_nm != start_nm:
            remainingNodes.append(node_nm)
    nodeAccWeight[start_nm] = 0
    nodeStatus[start_nm] = True
    selectedNodes.append(start_nm)
    #print("nodeAccWeight : ", nodeAccWeight)
    #print("Initial status of prevNode : ", prevNodes_nm)
```

```
# Graph with DFS, BFS, Dijkstra, MST(MinimumSpanningTree) (5)

    count = 1
    while len(remainingNodes) != 0:
        #print(">>> Round {} :".format(count))
        minAccWeight = PLUS_INF
        minNode = None
        #print(" -- currently selected {}, remaining {}“\
            .format(selectedNodes, remainingNodes))
        for n in remainingNodes:
            nAccWeight = nodeAccWeight[n]
            #print(" -- evaluating node ({:3}), nAccWeight({}) ...".format(n, nAccWeight))
            #print(" -- current minAccWeight = ", minAccWeight)
            if nAccWeight != None and nAccWeight < minAccWeight:
                minNode, minAccWeight = n, nodeAccWeight[n]
                #print("   ==> minAccWeight updated by newMinNode ({}) with minAccWeight
                  ({})".format(minNode, minAccWeight))
        if minNode == None:
            print("No minNode was selected at this round !!")
            print("Error - graph is not fully connected !!")
            errorInLoop = True
            break
        else:
            #print(" -- newly selected minNode : {}".format(minNode))
            selectedNodes.append(minNode)
            minAccWeight = nodeAccWeight[minNode]
            # edge relaxations
            for rn in remainingNodes:
                if rn == minNode:
                    continue
                e = Edge(minNode, rn)
                eWeight = G.getEdgeWeight(e)
                #print(" -- eWeight({}->{}):{}".format(minNode, rn, eWeight))
                if eWeight == None:
                    continue
```

```
# Graph with DFS, BFS, Dijkstra, MST(MinimumSpanningTree) (5)

                if nodeAccWeight[rn] > minAccWeight + eWeight:
                    nodeAccWeight[rn] = minAccWeight + eWeight
                    prevNodes_nm[rn] = minNode
                    #print(" -- Updated nodeAccWeight for node ({:3}) with prevNode
                        ({:3})".format(rn, minNode))
            if minNode == end_nm:
                # reached to destination
                break
            remainingNodes.remove(minNode)
        #print(" -- PrevNode : ", prevNodes_nm)
        #print(" -- Remaining nodes : ", end='')
        """
        for rn in remainingNodes:
            print("{} ({})".format(rn,nodeAccWeight[rn]), end=', ')
        print()
        """
        count += 1
    #
    if errorInLoop == True:
        return None

    print(" prevNode : ", prevNodes_nm)
    path = [end_nm]
    cur_node_nm = end_nm
    while cur_node_nm in selectedNodes:
        #print("Current path : ", path)
        if cur_node_nm == start_nm:
            break
        else:
            cur_node_nm = prevNodes_nm[cur_node_nm]
            path.insert(0,cur_node_nm)
            #print("Current path : ", path)
    return path, nodeAccWeight[end_nm]
```

```
# Graph with DFS, BFS, Dijkstra, MST(MinimumSpanningTree) (5)

def initGraph(G):
    node_names = ["SF", "LA", "DLS", "CHG", "MIA", "NY", "BOS"]
    w_edges = [WeightedEdge("SF", "LA", 337), WeightedEdge("LA", "SF", 337),\
               WeightedEdge("SF", "DLS", 1464), WeightedEdge("DLS", "SF", 1464),\
               WeightedEdge("SF", "BOS", 2704), WeightedEdge("BOS", "SF", 2704),\
               WeightedEdge("SF", "CHG", 1846), WeightedEdge("CHG", "SF", 1846),\
               WeightedEdge("LA", "DLS", 1235), WeightedEdge("DLS", "LA", 1235),\
               WeightedEdge("LA", "MIA", 2342), WeightedEdge("MIA", "LA", 2342),\
               WeightedEdge("DLS", "MIA", 1121), WeightedEdge("MIA", "DLS", 1121),\
               WeightedEdge("DLS", "CHG", 802), WeightedEdge("CHG", "DLS", 802),\
               WeightedEdge("CHG", "NY", 740), WeightedEdge("NY", "CHG", 740),\
               WeightedEdge("CHG", "BOS", 867), WeightedEdge("BOS", "CHG", 867),\
               WeightedEdge("NY", "MIA", 1090), WeightedEdge("MIA", "NY", 1090),\
               WeightedEdge("NY", "BOS", 187), WeightedEdge("BOS", "NY", 187),\
               WeightedEdge("BOS", "MIA", 1258), WeightedEdge("MIA", "BOS", 1258) ]
    for i in range(len(node_names)):
        v_name = node_names[i]
        node = Node(v_name)
        print("initGraph() :: adding node ({}) into Graph".format(node.getName()))
        G.addNode(node)
    for we in w_edges:
        #print("\ninitGraph() :: adding weighted_edge {} into Graph".format(we))
        G.addEdge(we)
    return G

def searchSP_DFS(graph, start_nm, end_nm):
    return DFS(graph, start_nm, end_nm, [], None)

def searchSP_BFS(graph, start_nm, end_nm):
    return BFS(graph, start_nm, end_nm)

def searchSP_Dijkstra(graph, start_nm, end_nm):
    return Dijkstra(graph, start_nm, end_nm)
```

```
# Graph with DFS, BFS, Dijkstra, MST(MinimumSpanningTree) (5)

if __name__ == "__main__":
    G = WeightedGraph ()
    initGraph(G)
    node_names = G.getNode_NMs()
    print("Nodes : ", node_names)
    print("Edges :")
    G.printEdges()
    print("\nConnectivity:")
    G.printConnectivity()
```

```
initGraph() :: adding node (SF) into Graph
initGraph() :: adding node (LA) into Graph
initGraph() :: adding node (DLS) into Graph
initGraph() :: adding node (CHG) into Graph
initGraph() :: adding node (MIA) into Graph
initGraph() :: adding node (NY) into Graph
initGraph() :: adding node (BOS) into Graph
Nodes :  ['SF', 'LA', 'DLS', 'CHG', 'MIA', 'NY', 'BOS']
Edges :
  SF ->(337)->LA,   LA ->(337)->SF,   SF ->(1464)->DLS,   DLS->(1464)->SF,   SF ->(2704)->BOS,
  BOS->(2704)->SF,   SF ->(1846)->CHG,   CHG->(1846)->SF,   LA ->(1235)->DLS,   DLS->(1235)->LA,
  LA ->(2342)->MIA,   MIA->(2342)->LA,   DLS->(1121)->MIA,   MIA->(1121)->DLS,   DLS->(802)->CHG,
  CHG->(802)->DLS,   CHG->(740)->NY,   NY ->(740)->CHG,   CHG->(867)->BOS,   BOS->(867)->CHG,
  NY ->(1090)->MIA,   MIA->(1090)->NY,   NY ->(187)->BOS,   BOS->(187)->NY,   BOS->(1258)->MIA,
  MIA->(1258)->BOS,
Connectivity:
 AdjacencyList[SF] = ['LA', 'DLS', 'BOS', 'CHG']
 AdjacencyList[LA] = ['SF', 'DLS', 'MIA']
 AdjacencyList[DLS] = ['SF', 'LA', 'MIA', 'CHG']
 AdjacencyList[CHG] = ['SF', 'DLS', 'NY', 'BOS']
 AdjacencyList[MIA] = ['LA', 'DLS', 'NY', 'BOS']
 AdjacencyList[NY] = ['CHG', 'MIA', 'BOS']
 AdjacencyList[BOS] = ['SF', 'CHG', 'NY', 'MIA']
```

```
# Graph with DFS, BFS, Dijkstra, MST(MinimumSpanningTree) (6)

    #print("weightedEdges : ", G.getWeightedEdges())
    start_nm = "LA"
    end_nm = "BOS"
    print("\nTrying DFS : ({} -> {})".format(start_nm, end_nm))
    path_DFS = searchSP_DFS(G, start_nm, end_nm)
    print("Path ({} -> {}) found by DFS : {}".format(start_nm, end_nm, path_DFS))

    print("\nTrying BFS : ({} -> {})".format(start_nm, end_nm))
    path_BFS = searchSP_BFS(G, start_nm, end_nm)
    print("Path ({} -> {}) found by BFS : {}".format(start_nm, end_nm, path_BFS))

    print("\nTrying ShortestPath_Dijkstra : ({} -> {})".format(start_nm, end_nm))
    path_Dijkstra, path_cost = searchSP_Dijkstra(G, start_nm, end_nm)
    print("ShortestPath_Dijkstra ({} -> {}): {}, path_cost ={}".format(start_nm,
        end_nm, path_Dijkstra, path_cost))
```

```
Trying DFS : (LA -> BOS)
Path (LA -> BOS) found by DFS : ['LA', 'SF', 'DLS', 'MIA', 'NY', 'CHG', 'BOS']

Trying BFS : (LA -> BOS)
Path (LA -> BOS) found by BFS : ['LA', 'SF', 'BOS']

Trying ShortestPath_Dijkstra : (LA -> BOS)
 Initial weight of edge (LA ->SF ) = 337:
 Initial weight of edge (LA ->LA ) = 0:
 Initial weight of edge (LA ->DLS) = 1235:
 Initial weight of edge (LA ->CHG) = 2147483647:
 Initial weight of edge (LA ->MIA) = 2342:
 Initial weight of edge (LA ->NY ) = 2147483647:
 Initial weight of edge (LA ->BOS) = 2147483647:
 prevNode :  {'SF': 'LA', 'LA': 'LA', 'DLS': 'LA', 'CHG': 'DLS', 'MIA': 'LA', 'NY': 'CHG', 'BOS': 'CHG'}
ShortestPath_Dijkstra (LA -> BOS): ['LA', 'DLS', 'CHG', 'BOS'], path_cost =2904
```

Homework 11

Homework 11

11.1 정수형 난수 리스트 정렬

- 지정된 크기의 중복되지 않는 정수형 난수 리스트를 생성하여 반환하는 함수 genRandList(L, L_size)와 큰 크기의 리스트의 첫부분과 끝부분을 샘플로 출력하는 함수 printListSample(L, per_line, sample_lines)를 포함하는 사용자 정의 모듈 myList.py를 구현하라.
- 주어진 리스트의 원소들을 정렬하기 위한 선택정렬 함수 selectionSort(), 병합정렬 함수 mergeSort(), 퀵정렬함수 quicksort()를 사용자 정의 모듈 mySortings.py에 직접 구현하라.
- 응용 프로그램에서는 myList 모듈과 mySortings 모듈을 import하여 사용할 수 있도록 준비하고, main() 함수에서는 정수형 난수 리스트의 크기를 입력받고, 중복되지 않는 난수 리스트를 selectionSort(), mergeSort(), quickSort() 함수를 사용하여 각각 정렬할 때 걸리는 시간을 파이썬 time 모듈을 사용하여 측정 및 출력하고, 성능을 비교하라. 단, selectionSort() 함수는 50000이하의 크기에만 실행시킬 것.
- 이 정렬 함수들의 성능에 차이가 나는 이유에 대하여 설명하라..
- main() 함수 (예시) : 다음 페이지 참조
- 실행결과 예시 : 다음 페이지 참조

```
# Homework 11.1 Performance Comparisons of Sorting Algorithms (1)

import os, sys, random, time
import myList, mySortings

def main():
    Selection_Sort_Limit = 50000
    while True:
        L = []
        print("\nComparisons of sorting algorithms")
        L_Size = int(input("Input array/list Size (0 to quit) = "))
        if L_Size == 0:
            break
        myList.genRandList(L, L_Size)

        # Testing Selection Sorting
        if L_Size <= Selection_Sort_Limit:
            random.shuffle(L)
            print("\nBefore Selection-Sort of L :")
            myList.printListSample(L, 10, 3)
            t1 = time.time()
            mySortings.selectionSort(L)
            t2 = time.time()
            print("After Selection-Sort of L .....")
            myList.printListSample(L, 10, 3)
            time_elapsed = t2 - t1
            print("Selection sorting took {} sec".format(time_elapsed))
```

```
# Homework 11.1 Performance Comparisons of Sorting Algorithms (2)

        # testing MergeSorting
        random.shuffle(L)
        print("\nBefore mergeSort of L :")
        myList.printListSample(L, 10, 3)
        t1 = time.time()
        L_sorted = mySortings.mergeSort(L)
        t2 = time.time()
        print("After mergeSort of L :")
        myList.printListSample(L_sorted, 10, 3)
        time_elapsed = t2 - t1
        print("Merge sorting took {} sec".format(time_elapsed))

        # testing Quick Sorting
        random.shuffle(L)
        print("\nBefore quickSort of L :")
        myList.printListSample(L, 10, 3)
        t1 = time.time()
        mySortings.quickSort(L)
        t2 = time.time()
        print("After quickSort of A :")
        myList.printListSample(L, 10, 3)
        time_elapsed = t2 - t1
        print("Quick sorting took {} sec".format(time_elapsed))

if __name__ == "__main__":
    main()
```

```
Comparisons of sorting algorithms
Input array Size (-1 to quit) = 30000

Before Selection-Sort of A :
   5795   22216    3791   26638   11938   14470   24802    8274   26053   28367
   3346    8088   14767   27633   28168   12821    3195   20076   27493    9903
  26943   25912   26364   29945   21855   17591   18414    5256   14150    4303
. . . . . .
  13044   19456   28267   20800   21066   15349   28446   19472   29080    9754
   5667   24757    3349    3180    7558   10718   10968   25307   25185    5064
  22544   22100    8711    7883    2192    7774    8549    2057     515   29848
After Selection-Sort of A .....
      0       1       2       3       4       5       6       7       8       9
     10      11      12      13      14      15      16      17      18      19
     20      21      22      23      24      25      26      27      28      29
. . . . . .
  29970   29971   29972   29973   29974   29975   29976   29977   29978   29979
  29980   29981   29982   29983   29984   29985   29986   29987   29988   29989
  29990   29991   29992   29993   29994   29995   29996   29997   29998   29999
Selection sorting took 28.802119255065918 sec

Before mergeSort of A :
  20753   26231   24213   10243   23819   16176    8846   24526   11008    3718
  12871   29273   10233   21945   14221   11630    5944   26235    6115   23796
   3711    1005   12323    7163   17683    2595   29794   29682   14317   11716
. . . . . .
   8355   18834   22339    2872    1063   29613   12199    7783   18795   10461
  22466    1094   19854   21051   13951   18632   24113   27294    7104     329
   3173    8432   22000   27700   25344   26573    7725    7212   11665   19647
After mergeSort of A :
      0       1       2       3       4       5       6       7       8       9
     10      11      12      13      14      15      16      17      18      19
     20      21      22      23      24      25      26      27      28      29
. . . . . .
  29970   29971   29972   29973   29974   29975   29976   29977   29978   29979
  29980   29981   29982   29983   29984   29985   29986   29987   29988   29989
  29990   29991   29992   29993   29994   29995   29996   29997   29998   29999
Merge sorting took 0.09175896644592285 sec

Before quickSort of A :
  24431   11388   18154   18951    4281   15949   21740    2976    5921   25025
  27213   23142   26137   15519    6810    7352   22763   29634    5108    2622
  23891   23080    8721   19892   18307   23766    1969   25685   27589   20490
. . . . . .
  22794   26781   21505   26035    5878   28438   11521    9348   11552   13885
  22561   15609    3120   12919   21003   23264   27754   28522   26665   15869
   3386   21271    1113   25212    1137    3600   23229   20534   28019   14264
After quickSort of A :
      0       1       2       3       4       5       6       7       8       9
     10      11      12      13      14      15      16      17      18      19
     20      21      22      23      24      25      26      27      28      29
. . . . . .
  29970   29971   29972   29973   29974   29975   29976   29977   29978   29979
  29980   29981   29982   29983   29984   29985   29986   29987   29988   29989
  29990   29991   29992   29993   29994   29995   29996   29997   29998   29999
Quick sorting took 0.0728304386138916 sec
```

```
Comparisons of sorting algorithms
Input array Size (-1 to quit) = 500000

Before mergeSort of A :
 451983  439385  335234  200446  377548  187175  182155   78922  273514  405024
 337679   16961  117773  280372  159829   33534   96853  440601  394714   17056
 109712  329627  353620   60948  109459  247068  300400  309808  265843   55482
. . . . . .
 288859  434439  481153  105425  127313  276477  299826  468955  143699  330204
 285156   91777  454380  167313   57925  243456  451574  269244  151963  484010
 257825  334507   29620  153673  304027  198708  388328  293327  353340   37851
After mergeSort of A :
      0       1       2       3       4       5       6       7       8       9
     10      11      12      13      14      15      16      17      18      19
     20      21      22      23      24      25      26      27      28      29
. . . . . .
 499970  499971  499972  499973  499974  499975  499976  499977  499978  499979
 499980  499981  499982  499983  499984  499985  499986  499987  499988  499989
 499990  499991  499992  499993  499994  499995  499996  499997  499998  499999
Merge sorting took 2.015639305114746 sec

Before quickSort of A :
 310603   70051   72047  383683  444732   96927  419898   63443    9447  363329
 479140  232056  385698   78024  110660   85868  257729  148928  220786  228990
 214143  191179   54730  473351  489079  192060  252417  472208  201591  497930
. . . . . .
 462447   74396  349189  261960    8516  192539   76763  313333  469481  300368
 180587  313432  230895  364747  200063   11880  345215  398586  491623  229930
 258181  345883  468096  149311  246623  153602  233000  273616  474748  380121
After quickSort of A :
      0       1       2       3       4       5       6       7       8       9
     10      11      12      13      14      15      16      17      18      19
     20      21      22      23      24      25      26      27      28      29
. . . . . .
 499970  499971  499972  499973  499974  499975  499976  499977  499978  499979
 499980  499981  499982  499983  499984  499985  499986  499987  499988  499989
 499990  499991  499992  499993  499994  499995  499996  499997  499998  499999
Quick sorting took 1.6854937076568604 sec
```

Homework 11.4

11.2 학생 정보 레코드의 HashMap 구성 및 검색

- 다양한 정보 레코드를 가변적인 길이의 문자열인 키 (key)를 사용하여 신속하게 검색할 수 있게 하는 class Entry(), class Bucket(Entry), class HashMap(Bucket)을 사용자 정의 모듈 myHashMap.py에 구현하라. 인수로 주어진 가변길이 문자열 str에 대한 해시값 계산을 위한 함수 CyclicShiftHashCode(str_key)를 함께 구현할 것.
- 응용 프로그램의 main() 함수에서는 학생에 대한 정보 레코드 (이름 (5 ~ 10 영문 문자), 학번, 학과, 성적(GPA))를 학생 이름을 key로 사용하여 해당 학생 정보를 신속하게 검색할 수 있는 해시맵 (hash map)을 구현하라. 10명의 학생 정보를 포함하는 리스트 L_students로 준비하고, 직접 구현한 해시맵에 포함시킨 후, 학생 이름을 입력받아 해당 학생 정보를 검색하고 출력하는 프로그램을 작성하라.
- HashMap은 7개의 bucket을 사용하도록 구성할 것.
- L_students에 포함된 학생 이름 5개와 포함되지 않는 학생 이름 1개에 대한 실행 결과를 출력하라.
- main() 함수 예시 : 다음 페이지 참조
- 실행 결과 예시 : 다음 페이지 참조

```
# Homework 11.2 Application of HashMap for Student Records

import myHashMap

def main():
    HashMap_Capacity = 7
    print("Creating a HashMap of capacity ({})".format(HashMap_Capacity))
    hsMap = myHashMap.HashMap(capacity=HashMap_Capacity)
    Entries = [("Kim", 19345, "ICE", 4.0), ("Park", 18234, "CS", 4.2), ("Hong", 20456, "EE", 3.9),\
               ("Lee", 20987, "ME", 3.8), ("Yoon", 21654, "ICE", 3.7), ("Moon", 21001, "CHEM", 4.1),\
               ("Hwang", 21123, "CE", 3.7), ("Choi", 19003, "EE", 4.3), ("Yeo", 20234, "ME", 3.8),\
               ("Jeong", 18005, "PH", 4.3)]
    for i in range(len(Entries)):
        entry = Entries[i]
        key = entry[0]
        hsMap._setitem(key, entry)
        print("Entry[{:2}] : {}".format(i, Entries[i]))
    print("Current HashMap Internal Structure:\n", hsMap)

    print("Checking entry searching in HashMap")
    while True:
        key = input("Input student name to search (. to quit) : ")
        if key == '.':
            break
        v = hsMap._getitem(key)
        if v == None:
            print("key ({}) is not found in hashmap !!".format(key))
        else:
            print("key ({}) : Value ({})".format(key, v))


if __name__ == "__main__":
    main()
```

```
Creating a HashMap of capacity (7)
Entry[ 0] : ('Kim', 19345, 'ICE', 4.0)
Entry[ 1] : ('Park', 18234, 'CS', 4.2)
Entry[ 2] : ('Hong', 20456, 'EE', 3.9)
Entry[ 3] : ('Lee', 20987, 'ME', 3.8)
Entry[ 4] : ('Yoon', 21654, 'ICE', 3.7)
Entry[ 5] : ('Moon', 21001, 'CHEM', 4.1)
Entry[ 6] : ('Hwang', 21123, 'CE', 3.7)
Entry[ 7] : ('Choi', 19003, 'EE', 4.3)
Entry[ 8] : ('Yeo', 20234, 'ME', 3.8)
Entry[ 9] : ('Jeong', 18005, 'PH', 4.3)
Current HashMap Internal Structure:
 bucket[ 0] : Kim, Yeo, Jeong,
 bucket[ 3] : Hwang,
 bucket[ 4] : Park, Hong, Yoon,
 bucket[ 5] : Choi,
 bucket[ 6] : Lee, Moon,
```

```
Checking entry searching in HashMap
Input student name to search (. to quit) : Yeo
key(Yeo) => hash_tbl[0]
key (Yeo) : Value (('Yeo', 20234, 'ME', 3.8))
Input student name to search (. to quit) : AAA
key(AAA) => hash_tbl[0]
key (AAA) is not found in hashmap !!
Input student name to search (. to quit) : Hong
key(Hong) => hash_tbl[4]
key (Hong) : Value (('Hong', 20456, 'EE', 3.9))
Input student name to search (. to quit) : Yoon
key(Yoon) => hash_tbl[4]
key (Yoon) : Value (('Yoon', 21654, 'ICE', 3.7))
Input student name to search (. to quit) : Moon
key(Moon) => hash_tbl[6]
key (Moon) : Value (('Moon', 21001, 'CHEM', 4.1))
Input student name to search (. to quit) : BBB
key(BBB) => hash_tbl[0]
key (BBB) is not found in hashmap !!
Input student name to search (. to quit) : .
```

Homework 11.5

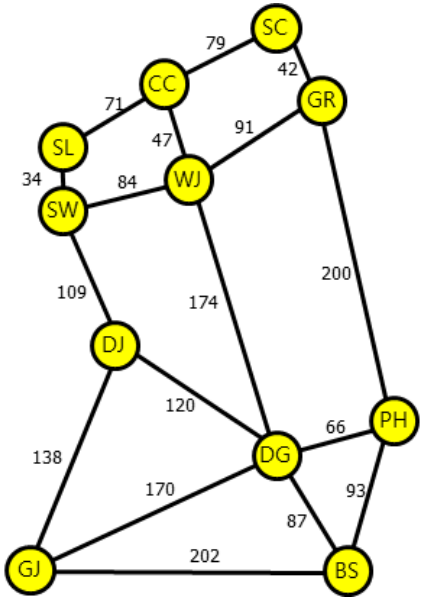
11.3 Dijkstra 알고리즘을 사용한 최단 거리 경로 탐색

(1) 요구사항

- 그래프 자료구조와 Dijkstra 알고리즘을 기반으로 하여 주어진 지도상의 시작 위치와 목적지 위치간의 최단 거리 경로를 찾아 출력하는 파이썬 프로그램을 작성하라.
- 그래프 자료구조를 구현하기 위하여 파이썬 클래스 class Node, class Edge, class WeightedEdge, class WeightedGraph를 구현하라.
- 그래프 자료구조 기반으로 WeightedGraph에 포함된 노드들 간의 최단 거리 경로를 찾아 출력하도록 파이썬 함수 Dijkstra(G, start_nm, end_nm)을 작성하라.

(2) 그래프 토폴로지 및 최단거리 경로 탐색

- 다음은 한국의 11개 도시를 연결하는 그래프이며, 17개의 간선 (edge)에 표시된 숫자는 거리 (distance)를 나타낸다.
- 그래프의 노드와 간선을 추가하도록 파이썬 함수 initGraph()을 작성하라.
- 이 그래프에서 주어진 출발도시에서 도착도시까지의 최단거리 경로를 찾아 출력하는 파이썬 응용 프로그램을 구성하고, GJ --> SC, SC --> GJ, SL --> BS, BS --> SL 간의 최단거리 경로를 각각 찾아 해당 경로의 누적 거리 (path_cost)와 함께 출력하라.



(3) main() 함수 (예시)

```
def main():
    G = Digraph()
    initGraph(G) # inserts nodes and edges into Digraph G
    nodes = G.getNode_NMs()
    print("Nodes : ", nodes)
    edges = G.getWEdges()
    print("Edges :")
    G.printEdges() # use printEdges() method in class Digraph
    print("Connectivity :")
    G.printConnectivity()
    #print("₩nTrying ShortestPath_Dijkstra : ({} -> {})".format("GJ", "SC"))
    path_Dijkstra, path_cost = Dijkstra(G, "GJ", "SC")
    print("ShortestPath_Dijkstra ({} -> {}): {}, path_cost ={}"₩
      .format("GJ", "SC", path_Dijkstra, path_cost))
    path_Dijkstra, path_cost = Dijkstra(G, "SC", "GJ")
    print("ShortestPath_Dijkstra ({} -> {}): {}, path_cost ={}"₩
      .format("SC", "GJ", path_Dijkstra, path_cost))
    #print("₩nTrying ShortestPath_Dijkstra : ({} -> {})".format("SL", "BS"))
    path_Dijkstra, path_cost = Dijkstra(G, "SL", "BS")
    print("ShortestPath_Dijkstra ({} -> {}): {}, path_cost ={}"₩
      .format("SL", "BS", path_Dijkstra, path_cost))
    path_Dijkstra, path_cost = Dijkstra(G, "BS", "SL")
    print("ShortestPath_Dijkstra ({} -> {}): {}, path_cost ={}"₩
      .format("BS", "SL", path_Dijkstra, path_cost))
if __name__ == "__main__":
    main()
```

(4) 실행결과 (예시):

```
Nodes :  ['SL', 'CC', 'SC', 'SW', 'WJ', 'GR', 'DJ', 'GJ', 'DG', 'PH', 'BS']
Edges :
  SL->CC ( 71),   CC->SL ( 71),   SL->SW ( 34),   SW->SL ( 34),   CC->SC ( 79),
  SC->CC ( 79),   CC->WJ ( 47),   WJ->CC ( 47),   SC->GR ( 42),   GR->SC ( 42),
  SW->WJ ( 84),   WJ->SW ( 84),   WJ->GR ( 91),   GR->WJ ( 91),   GR->PH (200),
  PH->GR (200),   SW->DJ (109),   DJ->SW (109),   DJ->GJ (138),   GJ->DJ (138),
  GJ->DG (170),   DG->GJ (170),   WJ->DG (174),   DG->WJ (174),   DG->DJ (120),
  DJ->DG (120),   DG->PH ( 66),   PH->DG ( 66),   DG->BS ( 87),   BS->DG ( 87),
  GJ->BS (202),   BS->GJ (202),   PH->BS ( 93),   BS->PH ( 93),
Connectivity:
 AdjacencyList[SL] = ['CC', 'SW']
 AdjacencyList[CC] = ['SL', 'SC', 'WJ']
 AdjacencyList[SC] = ['CC', 'GR']
 AdjacencyList[SW] = ['SL', 'WJ', 'DJ']
 AdjacencyList[WJ] = ['CC', 'SW', 'GR', 'DG']
 AdjacencyList[GR] = ['SC', 'WJ', 'PH']
 AdjacencyList[DJ] = ['SW', 'GJ', 'DG']
 AdjacencyList[GJ] = ['DJ', 'DG', 'BS']
 AdjacencyList[DG] = ['GJ', 'WJ', 'DJ', 'PH', 'BS']
 AdjacencyList[PH] = ['GR', 'DG', 'BS']
 AdjacencyList[BS] = ['DG', 'GJ', 'PH']
ShortestPath_Dijkstra (GJ -> SC): ['GJ', 'DJ', 'SW', 'SL', 'CC', 'SC'], path_cost =431
ShortestPath_Dijkstra (SC -> GJ): ['SC', 'CC', 'SL', 'SW', 'DJ', 'GJ'], path_cost =431
ShortestPath_Dijkstra (SL -> BS): ['SL', 'SW', 'DJ', 'DG', 'BS'], path_cost =350
ShortestPath_Dijkstra (BS -> SL): ['BS', 'DG', 'DJ', 'SW', 'SL'], path_cost =350
```