Lecture 13 : 리스트 자료구조의 성능 평가

컨테이너 자료구조의 대표적인 배열과 리스트 중에서 선택의 기준은 사용할 동작의 효율이다. 효율은 크게 시간 복잡도 (time complexity)와 공간 복잡도 (space complexity)'로 평가된다. 그중에서도 더 중요한 것은 시간 복잡도이며 이것은 실제 수행시간으로 확인될 수 있다.

자료구조는 크게 정적인 것과 동적인(dynamic) 것으로 대별된다. 자료구조가 동적이라고 하는 것은 삽입과 삭제 등으로 전체 자료구조의 물리적 크기나 형상이 변화하는 것을 의미한다. 우리는 삽입 삭제 정도만 고려하면 된다. 탐색과 데이터 fetch에 매우 효율적인 배열구조는 삽입과 삭제에 상당히 취약하다. 특히 정렬된 상태에서 새로운 자료가 들어오면 그 순서를 유지하기 위하여 전체의 위치를 이동시켜야 한다. 정렬된 배열에서 가장 극단적인 경우는 삽입된 원소가 제일 작은 경우, 즉 배열의 첫 위치에 놓이는 경우이다. 이 경우 이미 저장된 모든 원소는 한 칸씩 뒤로 물러나야 하므로 만일 저장된 원소가 N개라면 $O(N)$번의 data movement가 필요하다.

리스트는 이런 동적인 자료구조를 지원하는 가장 일반적인 자료구조이다. 단 사용자는 물리적인 위치(address)가 아닌 pointer 정보를 따로 관리해야 하는 부가적인 작업을 해야 한다. 그러나 STL에서 제공하는 list container를 사용하면 이런 복잡한 포인터 처리 문제를 깔끔하게 지원받을 수 있기 때문에 여러분은 STL list에서 제공해주는 다양한 member function을 실제 문제에서 활용하여 사용할 수 있어야 하고 이것을 현장에서 고민 없이 바로 활용할 수 있도록 충분한 연습을 해두어야 할 것이다.

동적인 자료구조라고 해서 항상 리스트를 사용해야 하는 것은 아니다. 만일 배열이나 vector를 사용해서도 실제 수행시간 상으로 크게 문제가 되지 않는다면 배열을 사용하는 것이 편리하다.[1] 따라서 여러분은 코드 전체 또는 특정 함수의 수행시간이 얼마나 걸리는지 측정(instrumentation)할 수 있어야 하고 또는 시스템에 내장된 profiling 도구를 자유롭게 활용하여 hot spot[2]을 찾아낸 다음 그 부분부터 집중적으로 개선을 해야 한다. 아래는 가장 일반적인 시간 측정 방법이다.

C++ 기반의 수행시간 측정 방법과 그 예

1) 배열과 벡터는 한꺼번에 전체 내용을 확인(출력)할 수 있어서 내장된 또는 전문적인 debugging tool을 활용하기에 유리하다.

2) 전체 수행에서 가장 시간을 많이 잡아먹는 코드의 일부. 이것을 확인하는 것이 성능을 개선하는 첫걸음이다. 이 hot spot을 확인하지 못하면 아무리 고생해도 성능을 개선시킬 수 없다. 단순한 코딩 technique 몇 개를 사용한다고 해도 전체 성능 변화는 미미하기 때문이다.

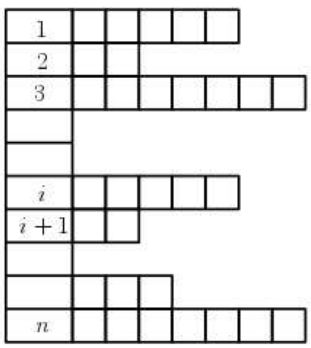
```cpp
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <chrono>

using namespace std;
using namespace chrono;

using intvec = vector<int>;
using intlist = list<int>;
using in_secs = duration<double>;

// 수행시간 비교하기

int main() {
    system_clock time;
    in_secs diff_time;
    int N = 200000 ;

    auto tbegin = time.now();
    intvec ex1{1,2,3};
    for(int i=0; i< N ; ++i)
        ex1.insert(ex1.begin(), i);
    auto tend = time.now();
    diff_time = tend - tbegin;
    cout << "Number of Steps = " << N << endl ;
    cout << "Time for vector.begin() : " << diff_time.count() <<
endl;

    tbegin = time.now();
    for(int i=0; i< N ; ++i)
        ex1.push_back(i);
    tend = time.now();
    diff_time = tend - tbegin;
    cout << "Time for vector.back()  : " << diff_time.count() <<
endl;


    tbegin = time.now();
    intlist ex2{1,2,3};
    for(int i=0; i< N ; ++i)
        ex2.insert(ex2.begin(), i);
    tend = time.now();
    diff_time = tend - tbegin;
    cout << "Time for list.begin() : " << diff_time.count() << endl;


    tbegin = time.now();
    for(int i=0; i< N ; ++i)
        ex2.push_back(i);
    tend = time.now();
    diff_time = tend - tbegin;
    cout << "Time for list.back() : " << diff_time.count()  << endl;

    return 0;

} // end of main()
```

Python의 경우

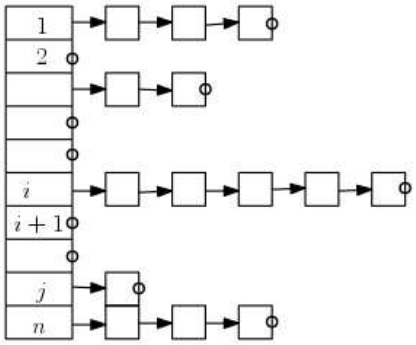
```
# time.clock( )은 더이상 지원되지 않습니다. 2021년 3월 봄날에

import time

N = 1000000

start = time.process_time()
for i in range( N ) :  # 시간을 측정하고자 하는 code
    a = 109813.0*1023801.2*i

delta = time.process_time() - start     #
print("Ellasped =", delta, " milli." )

for i in range( N ) :
    a = 109813.0*1023801.2*i
    b = a*a*a

delta = time.process_time() - start     # end
print("Ellasped =", delta, " milli." )
```

13.1 다음 4가지 자료구조를 설명하고 각 자료구조가 어떤 경우에 얼마나 더 효율적인지 구체적인 사례를 들어 설명하시오. (예, 각 학생별 출석 수업시간 저장)

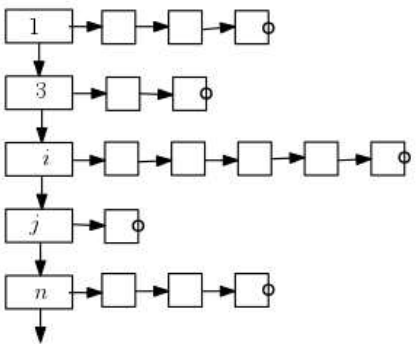
$vector < vector < int >> VV;$



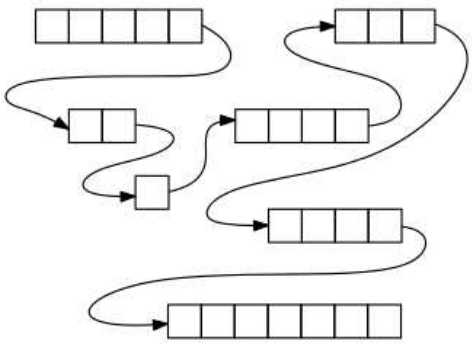
$vector < list < int >> VL;$



$list < list < int >> LL;$



$list < vector < int >> LV;$



13.2 2차원 sensor data를 3가지 방식으로 구성하는 방법으로 실제 구현을 생각해보자.
n개의 sensor가 있고 각 sensor는 매초 특정한 사건이 발생하였는지 기록한다.
사건은 대략 1/1000 확률로 발생한다. 이를 위한 3가지 자료구조가 다음과 같다.

- 먼저 3,000개의 random integer sequence를 container에 담습니다.
- 적어도 30번 이상 반복 시행을 한 뒤에 시간을 측정해야 합니다.
- 단 반복 시행에는 데이터를 다르게 해서 compiler가 optimization을 못하도록 해야 합니다. (시간 측정이 잘 안 될 수 있습니다.)

13.3 다음 표에 제시된 4개의 자료구조로 구현한 뒤 각각 수행시간을 측정해보시오.
단 데이터의 크기 N은 10만이 넘도록 해서 유의미한 시간이 나오도록 해야 한다.

측정값(sec.)	vector	deque	list	C (직접 구현)
head에 insert				
tail에 insert				
head에서 delete				
tail에서 delete				
중간에 insert				
모두 scan 하기				
전체 sorting				
find( )				
sort( )				
destruction( )				
reverse( )				

13.4 리스트의 원소에는 단일형 원소(int, float, string)이 들어갈 수도 있고, 복합형(struct 나 특정 object class)이 들어갈 수 있다. 실제로는 복합형이 많이 들어가게 됩니다.

a. `class student { int stud_id, string name, float grade }`이 선언되어 있을 때 각각의 id를 primary key의 내용으로 sort( ) 하는 방법을 찾아보시오.

b. 특정 원소가 리스트가 될 수 있다. 즉 list of list가 될 수 있다. 어떤 경우 ?
예) text는 문장의 연결리스트며, 각 문장의 단어의 연결리스트다.

c. 여러 문장이 순서대로 들어온다. 각 문장의 끝은 $로 표시되어 있다.
이들을 읽어서 리스트의 리스트로 만들어 보시오.

13.5 Python에서는 선형 컨테이너로 리스트와 배열을 제공한다. 리스트와 달리 배열은 homogeneous 한 원소만을 넣을 수 있어 random addressing의 속도를 높여준다. 실제 여러분의 컴퓨터 환경에서 Python array가 list보다 더 빠른지(append, insert(0), middle insert) 검사를 해보고 이것은 원소의 크기로 조사해보자.

13.5 다음은 어떤 컴퓨터의 메모리 맵이다. 여기에는 각각 한 자씩의 Character가 들어간다. 먼저 “computer” 라는 단어를 넣고 중간에 p를 제거해보라. 단 이 자료구조는 doubly linked list이기 때문에 중간 문자가 제거되어도 그 연결은 그대로 유지되어야 한다. 즉 “comuter" 라는 단어가 되어야 한다.

주소	tag	LLINK	DATA	RLINK
101				
102				
103			**R**	
104				
105				
106			**C**	
107				
108			**T**	
109				
110				
111				
112				
113			**P**	
114				
115				
116				
117				
118				
119				
120			**E**	
121				

주소	tag	LLINK	DATA	RLINK
101				
102				
103				
104				
105				
106			**U**	
107			**O**	
108				
109				
110				
111			**M**	
112				
113				
114				
115				
116				
117				
118				
119				
120				
121				

13.6 그리고 첫 'm' 다음에 또다시 새로운 'm'을 추가하라. "commuter"

13.7 Hard Disk Allocation에서 memory space를 관리하는 BUDDY SYSTEM을 설명하고 실제 아래 명령의 예로 실행해보시오. 명령의 예는 다음과 같다.

파일 추가 f는 이름 N은 크기	파일 f를 disk에서 제거	compact	report
write f N	Del f	Compact	디스크의 상황을 보고해야 한다.

전략 1) First-Fit
- 파일 F를 추가할 경우에는 앞에서 부터 연속된 빈 공간을 찾는다.
- 빈 공간에 F의 일부를 저장하고
- 다음 빈 공간에 F의 나머지 부분을 같은 방법으로 저장한다.

전략 2) Best-Fit
- 파일 F를 추가할 경우에는 앞에서 가장 큰 연속된 빈 공간을 찾는다.
- 만일 이러한 공간이 존재하면 채우고 작업을 종료한다.
- 만일 여유 공간이 없으면 가장 큰 연속 빈 공간에 넣을 수 있을 만큼 넣고
- F의 나머지를 같은 방법으로 저장한다.

Memory space가 100인 경우를 가정해서 다음 작업의 결과를 map으로 표시하라.

단계	작업	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
1	W X 40	X (40)																				
2	W Y 55																					
3	Del X																					
4	W R 15																					
5	W Q 25																					
6	Del Y																					
7	W M 35																					
8	Del R																					
9	W Z 15																					
10	Del Q																					
11	W U 25																					
12																						
13																						

다른 상황을 Best-Fit 방식으로 처리해 봅시다.

단계	작업	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
1	W A 40																					
2	W B 25																					
3	W R 15																					
4	Del B																					
5	W Q 30																					
6	Del R																					
7	W M 20																					
8	Del A																					
9	W Z 30																					
10	W Y 15																					
11	W U 25																					
12																						
13																						