Lecture 12 : 연결 리스트(linked list): 기초

이 장에서 이해해야 할 가장 중요한 개념은 자료의 물리적 순서와 논리적 순서의 차이(difference)다. 우리가 사용하는 컴퓨터의 기본적인 추상적 구조는 폰 노이만 구조에 기초하고 있어 기억장치는 선형적인, 즉 일렬로 늘어선 구조로 저장, 관리되고 있다. 따라서 모든 메모리는 하나의 index i로 결정된다. $i=0,1,2,...N$ 번지에 각각 특정한 원소를 저장할 수 있다. 그리고 특정 위치에 있는 원소를 가지고 오는 작업(fetch)하는 물리적인 위치에 상관없이 같다. 이것을 random access 구조라고 부른다. $i=0$번지에 있는 원소나 $i=2^{100}$번지에 있는 원소나 준비된 register로 옮기는 데 걸리는 시간은 같다.

실제 전자가 회로(wire)를 통과하는 속도는 광속과 같으므로 물리적으로 다른 거리에 있는 원소가 이동하는데 왜 같은 시간이 걸리는 것일까? 여기에는 약간의 "야로"[1]가 있다. 모든 컴퓨터는 system clock의 시간 단위로 동기를 하는데 그 단위 시간을 메모리에서 하나의 데이터가 register로 이동하는 데 걸리는 시간 중 가장 긴 시간을 단위로 하며 모든 데이터의 이동은 1 cycle time에 이동하는 것으로 되어 같게 된다.

이 경우 i번째 위치에 있는 원소와 $i+1$ 위치에 있는 원소는 물리적으로 "인접"해 있다고 표현한다. 원소가 물리적으로 선형적으로 인접해 있으면 원소를 찾는 데 매우 편리하다. 이런 예를 생각해보자. 우리는 어떤 town house[2]에서 특정 번지의 집(house)을 찾으려고 한다. 각 집(house)의 폭(width)은 같으므로 첫 집의 물리적 위치만 알고 있으면 i번째 집은 쉽게 찾을 수 있다. 즉 $i*w$(폭)만큼 이동하면 된다. 이것이 물리적으로 연속적인 데이터의 특징이다.

논리적으로 연결된 구조는 물리적으로 인접해 있지 않지만 어떤 의미 구조로 그 연결성이 보장되어 있다. 예를 들어 할아버지, 아버지, 자신, 아들[3], 이 4명의 혈연관계를 생각해보자. 이들은 각각 다른 장소, 광주, 부산, 대전, 제주도에 살고 있지만, 할아버지로부터 이어지는 그 위계는 보장되어 있다. 만일 각 사람의 자신의 1차 자손이 어디에 살고 있는지 알고 있다고 가정해보자. 그러면 할아버지를 통해서 아버지의 위치를 알고, 아버지를 통해서 자신, 그리고 당사자를 통해서 그 자식의 위치를 알아낼 수 있다. 이 경우 위 4명의 사람은 "논리적"으로 연결되어 있다고 표현을 한다.

물리적 인접 구조는 "주소"의 값으로 확인되지만, 논리적 연결구조는 위치 정보로 확인될 수 없으므로 각자 이웃의 정보를 따로 보관해야 하는 부담이 있다. 이것을 우리는

---

1) 마치 일본어같이 들리지만, 이것은 일본어가 아닌 순 우리 말이다. 이 뜻은 남에게 드러내지 않고 우물쭈물하는 셈속이나 수작을 속되게 이르는 말을 일컫는데. 다를 말로 하자면 흑막. 꼼수 이런 뜻이다. 예문 "그의 제안에는 분명히 무슨 ~가 있을 것이다. 조심해야 한다."

2) 일명 "집 장사"가 같은 모양으로 지은 집을 말한다. 같은 모양의 집을 일렬로 쭉 짓는다. 외국의 서민층이 거주하는 대표적인 거주구조이다.

3) 여성이라면 {할머니, 어머니, 자신, 딸}로 생각해도 된다.

pointer information이라고 부른다. 즉 서로 다른 위치에 존재하는 원소들이 추가의 pointer 정보를 이용해서 논리적으로 연결된 구조를 유지하는 것이다. 이것이 바로 논리적 연결구조이다.

우리가 어떤 Giga 단위의 동영상 파일을 볼 때(PC로 영화 볼 때를 생각해보면 된다) 별문제 없이 볼 수 있다. 그러나 실제 이 파일이 disc에 저장된 구조는 서로 떨어진 몇 개의 파일로 저장되어 있으며 각각은 논리적 연결구조로 연결되어 있다. 따라서 한 단위(segment)의 동영상 파일이 재생된 다음, disc의 다른 위치에 있는 파일을 다시 잽싸게 메모리에 올려서 재생시킨다. 물론 이 약간의 지체 현상은 우리가 느끼지 못할 수도 있지만 이러한 지체 현상이 우리가 인지할 정도가 되는 경우[4] 우리는 이것을 "버퍼링"이 일어난다고 말한다.

그러면 왜 이렇게 불편한 논리적 연결구조를 사용할까? 그 이유는 단순하다. 실제 남아있는 메모리 여유 공간의 합은 추가 기록된 파일의 크기보다 크지만 각각의 자투리 공간의 합은 기록될 파일의 크기보다 작을 경우이다. 예를 들어 크기 100인 파일이 준비되어 있지만 남아있는 공간은 40, 30, 50, 20, 30, 이렇게 총 170이 남아있는 경우이다. 이 경우에는 파일을 한쪽으로 몰아서 (compaction) 전체 자투리를 한 덩어리의 170 공간으로 만들 수 있지만, 이는 오랜 시간이 걸리는 작업이다. 이 경우 100 크기의 파일을 잘라서 각각 40, 30, 50(30)에 넣고 이를 논리적 구조로 연결하는 것이다.

연결 리스트(linked list)는 바로 이런 논리적 연속 구조를 지원하는 가장 기초적인 자료구조이며 현장에서 아주 많이 활용된다. 특히 데이터의 크기가 큰 경우, 데이터를 메인 메모리에 모두 담을 수 없어 보조 기억장치에 사용할 경우에는 반드시 사용해야 한다. 그런데 요즘은 SSD가 주 보조 장치로 활용되고 있어 새로운 방식의 저장 방법이 연구되고 있다.[5]

여러분이 이 장에서 배워야 할 사항은 언제 linked list가 사용되는가, 되어야 하는가, 실제 STL을 이용해서 linked list를 어떻게 사용해야 할 것인가? 그 과정에서 주의해야 할 사항은 무엇인가? 등이다.

가장 중요한 것은 실제 물리적인 연결구조인 array, vector와 linked list의 성능 차이를 실제 측정하여 이를 손끝에 "체화"시켜야 하는 것이다.

12.1 List는 그 자유스러움 때문에 현실에서 가장 많이 사용되는 자료구조다.

STL List 활용의 주요 Check Point

- 리스트의 원소에 다양한 데이터 타입을 넣을 수 있는가?
- 리스트를 원소로 하는 다차원 리스트를 만들 수 있는가?
- 리스트용 Overloading 함수를 만들 수 있는가?

---

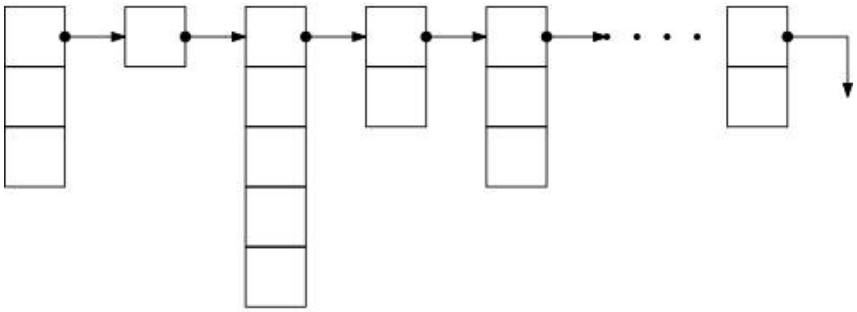
4) 사람의 경우 주파수에 따라서 그 차이를 인지하는 수준이 다르다고 한다.
https://www.newswise.com/articles/the-human-ear-detects-half-a-millisecond-delay-in-sound

5) SSD에서 메모리 compaction을 하는 것은 일반적인 회전형 disc에 비해서 매우 빠르다.

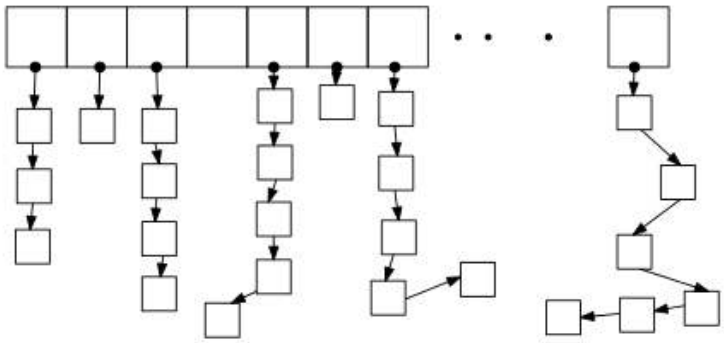
- 리스트에 부가 정보를 추가할 수 있는가? (중간원소, max 원소)
- 끝을 다시 처음에 연결하여 환형 리스트(circular list)를 만들 수 있는가?
- 리스트를 위하여 지원되는 다양한 member function을 사용할 수 있는가?
- 리스트에 적용되는 다양한 generic function (만능 함수)를 알고 있는가?
  예) `sort( anything ) ; <--> anything.sort( )`

아래는 vector와 list의 이중결합 형태의 모든 경우를 보여주고 있다. 어떤 문제에 다음의 자료구조를 사용하면 좋을지 생각해보자.

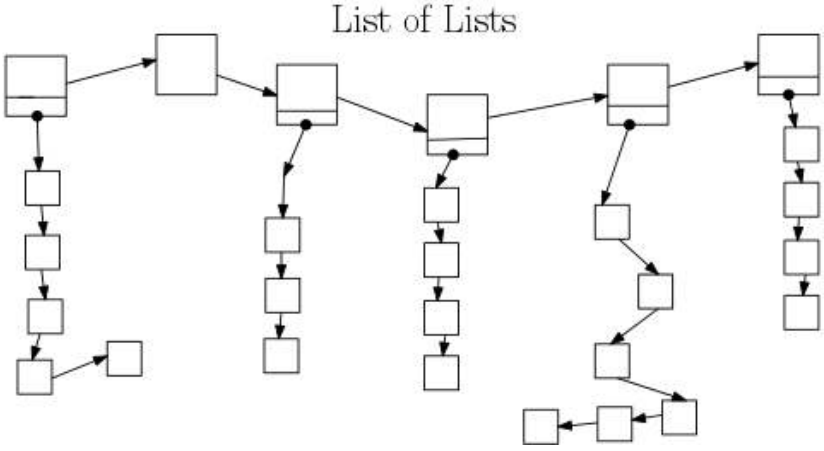
List of Vectors



Vector of Lists



List of Lists



vector : 정기적으로 발생하는 사건
list : 부정기적으로 발생하는 사건

- 비가 오는 수업일 중에서 결석한 학생 (학생은 모두 60명)

12.2 물리적 구조의 한계를 논리적 구조로 바꾼 자료구조

Q. 논리적으로 "가까운" 것과 물리적으로 "가까운" 것의 차이를 생각해보자.

a. 지하철 승객 칸에서 A와 B는 가까운 사이다.

b. 리스트 원소의 최소구조 : <Data, { link } >

c. 리스트의 물리적 구조와 논리적 구조

d. Connected World :- 세상은 <원소>와 그 <연결>로 이루어져 있다.

12.3 연결 리스트를 활용한 다양한 예

a. 비상 연락망

b. Hard-disk space 관리 (rather than main memory 관리)

c. 이상한 노트 필기 법

d. text editor

e. 다방에서 옛날식으로 친구 만나기 [6](아주 옛날, 휴대폰이 없는 시절)

12.4 연결 리스트의 다양한 형식 (이 모두는 STL의 list iterator로 처리할 수 있다.)

- 연결 방향에 따라서

단일 연결리스트 - 이중 연결리스트

- 연결 강도(connectivity)에 따라서

단순 연결리스트 - 다중 연결리스트

- 각 기본노드의 균일성에 따라서
  - 단순노드
  - 복합노드
- 단일 환형 리스트 (singly linked & circular list), 이중 환형리스트

Q) 왜 이런 다양한 형식의 linked list가 필요한 것일까요 ?

12.5 다양한 사용자 정의 리스트 :

text editor를 vector형 자료구조로 구현한다고 생각해보자. 무엇이 문제 ?

- insert, - delete

---

6) 1980년대 교수님의 대학 시절에는 폰이 없이 모임 장소를 바꿀 때는 이전 장소, 예를 들면 다방 입구에 늦게 오는 친구를 위해서 메모지를 항상 붙여 두었다. 예를 들면 "동수야, 우리는 여기 앞에 문창 당구장으로 간다. 9시 반까지 있을 예정이다."

12.6 대표적인 container 데이터 구조 중에서 vector와 list의 차이를 동작비용으로 이해하기

형식	특정 위치 삽입	특정 위치 삭제	임의 접근	삽입 후 반복자	메모리 사용량	사용 용도
vector						
list						
vector of vector						
list of list						
vector of list						
list of vector						

12.7 STL의 List 는 기본적으로 양방향 반복자(two way iterator)를 지원한다. 실제 구현은 doubly-linked list이다. 따라서 it+4 와 같은 작업은 불가능하다.

- list.insert (여기에, 이것을) ;
- list.erase (여기에)
- list.remove (이것을)
  list에 있는 "이것"과 일치하는 원소를 모두 지나면서 지움
- list.remove_if (Predicate)
  *predicate (프레디킷)은 결과가 bool={yes, no}인 함수

12.8 STL list로 할 수 있는 일

- 다양한 조건의 Linked list로 만드는 일 (예를 들면 circular list)
- 다양한 generic algorithm(기본으로 내장된 알고리즘) 사용하기
  - find( )
  - list.sort( )
  - list.unique( )
  - merge(lista, listb, operator) ;

12.9 STL list에서 발생되는 다양한 오류(exception)의 예과 대처방법(passing)

12.10 List용 다양한 iterator 직접 활용해보기

forward

backward

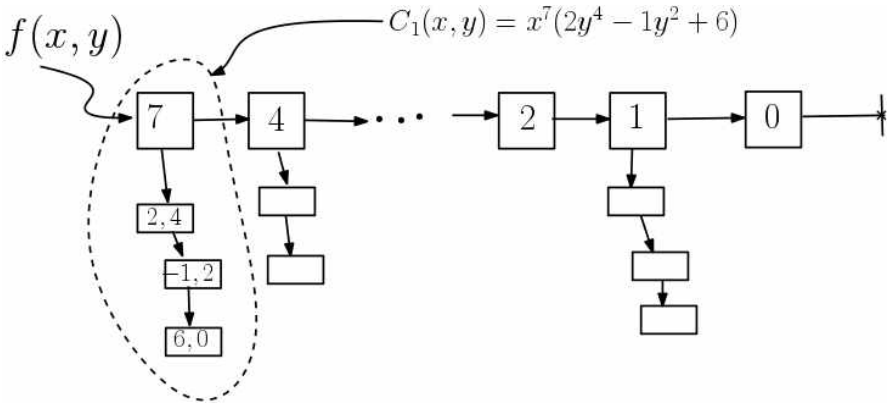
무한루핑

12.11 List를 응용한 실전 문제

- 2차 함수 표현하기

$$f(x,y) = x^3y^2 - 5x^3y - 2x^2y^4 + y^6 + xy^4 - 6x^7y^2 - 10x^5 - xy^2 + 1$$

$$g(x,y) = 5x^6y^2 + 6x^2y^7 - 4x^2 - y^4 + 5xy - x^6y^4 - 4x^2y^4 + 9$$



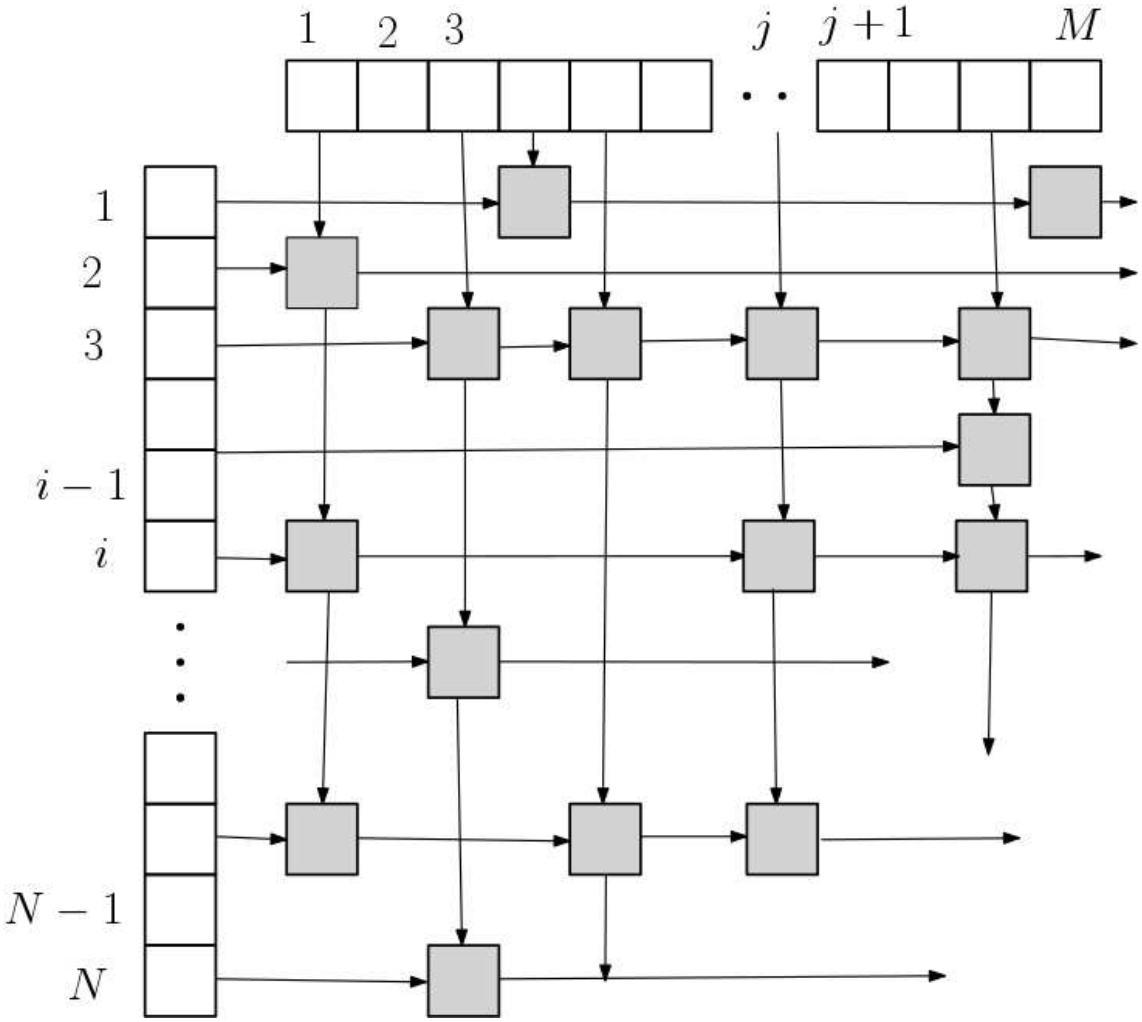
- 지원해야 할 동작(Operation)

$f(x^*, y^*)$, 모든 원소 출력

$f(x,y) + g(x,y)$, 함수 더하기

$f(x,y) \cdot g(x,y)$, 함수 곱하기

12.12 List를 응용한 실전 문제 : Sparse matrix 구현하기 N=10,000, M=10,000



지원해야 하는 동작(Operation)

a) w = M[i][j]

b) M[i][j] = z

c) print M[i][*]

d) print M[*][j]

e) M + W (matrix addition)

f) M * W (matrix product)