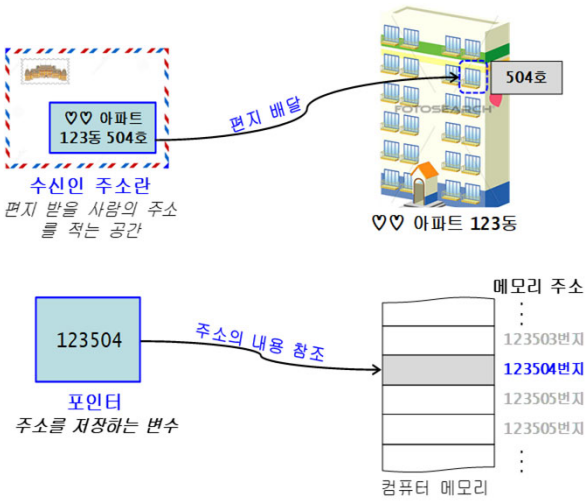
DATA
STRUCTURES
USING C

CHAPTER 05

포인터와 연결리스트

1

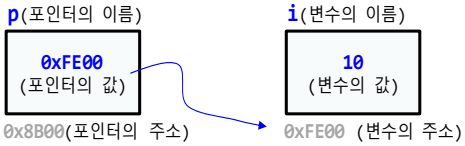
포인터(pointer)



2

포인터 선언

```
int i = 10;
int* p;
p = &i;
```



- 포인터 변수 선언

```
int* pi;       // int 변수를 가리킬 목적의 포인터 pi를 선언
float *pf;     // float 변수를 가리킬 목적의 포인터 pf를 선언
```

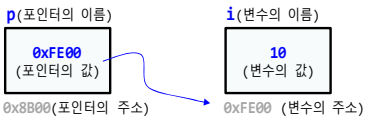
- 여러 개의 변수 선언

```
char* p, q, r;         // p는 char* 변수, q와 r은 char변수
char *p, *q, *r;       // p, q, r 모두 char*형 변수
```

- 사용하기 전에 반드시 초기화되어야 함

포인터 활용

```
int i = 10;
int* p;
p = &i;
```




```
*pi = 20;      // i=20과 동일
```

& 연산자
변수의 주소를 추출
* 연산자
포인터가 가리키는 곳의 내용을 추출

- * 연산자 사용의 여러가지 예

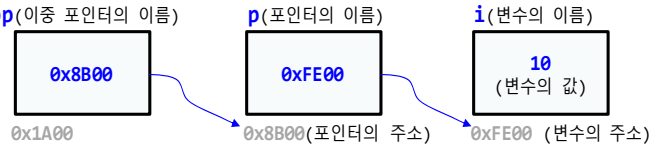
```
사용예#1:     c = a * b;
사용예#2:     int * p;
사용예#3:     *p = 20;
```

- 비트단위 AND 연산자 &

```
사용예#1:     c = a & b;
사용예#2:     p = &i;
사용예#3:     int & p;  // C++에서만 사용 가능(참조자)
```

이중 포인터

```
int i = 10;
int* p = &i;
int** pp = &p;
```



표현	자료형	동일한 표현
10	int (상수)	
i	int	*p, **pp
p	int*	*pp, &i
pp	int**	&p

5

포인터와 연산자

```
p           // 포인터
*p          // 포인터가 가리키는 값
*p++        // 가리키는 값을 가져온 다음, 포인터를 한칸 증가
*p--        // 가리키는 값을 가져온 다음, 포인터를 한칸 감소
(*p)++      // 포인터가 가리키는 값을 증가시킨다.
```

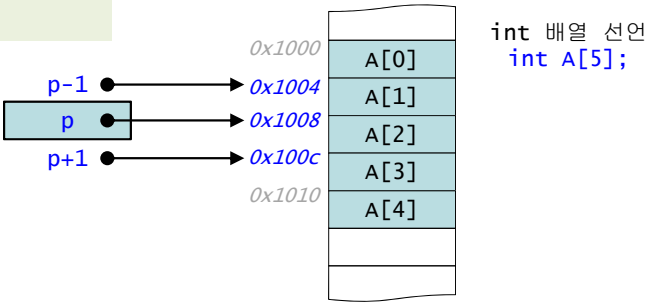
```
int a;      // 정수 변수 선언
int *p;     // 정수 포인터 선언
int **pp;   // 정수 포인터의 포인터 선언
p = &a;     // 변수 a와 포인터 p를 연결
pp = &p;    // 포인터 p와 포인터의 포인터 pp를 연결
```

6

포인터 연산

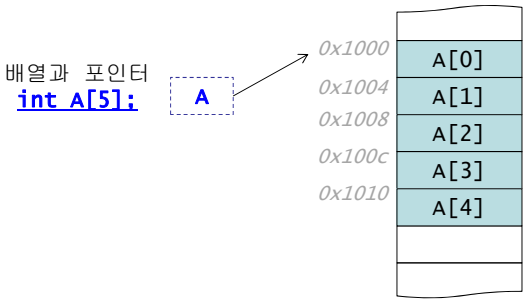
- 포인터에 대한 사칙연산
  - 포인터가 가리키는 객체단위로 계산된다.

```
int A[5], int *p;
p = &A[2];
p-1;
p+1;
```



배열과 포인터

- 배열의 이름: 사실상의 포인터와 같은 역할
  - 컴파일러가 배열의 이름을 배열의 첫 번째 주소로 대치



포인터와 구조체

```
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
} Polynomial ;
```

```
Polynomial a;
Polynomial* p;
p = &a;
a.degree = 5;
p->coef[0] = 1;
```

- 구조체 멤버의 접근
  - 구조체에서는 "." 연산자
  - 포인터에서는 "->" 연산자

구조체를 이용한 표현		포인터를 이용한 표현	
a.degree	(&a)->degree	p->degree	(*p).degree
a.coef[0]	(&a)->coef[0]	p->coef[0]	(*p).coef[0]

정적 메모리

- 정적 메모리 할당
  - 메모리의 크기는 프로그램이 시작하기 전에 결정
  - 실행 도중에 크기를 변경할 수 없다.
  - 더 큰 입력이 들어온다면 처리하지 못함
  - 더 작은 입력이 들어온다면 메모리 공간 낭비

```
int i;              // int형 변수 i를 정적으로 할당
int* p;             // 포인터 변수 p를 정적으로 할당
int A[10];          // 길이가 10인 배열을 정적으로 할당
```

동적 메모리 라이브러리 함수

- 실행 도중에 메모리를 할당 받는 것
  - 필요한 만큼만 할당을 받고 반납함
  - 메모리를 매우 효율적으로 사용가능

```
void* malloc(int size);
void* calloc(int num, int size);
```

```
void free(void* ptr)
```

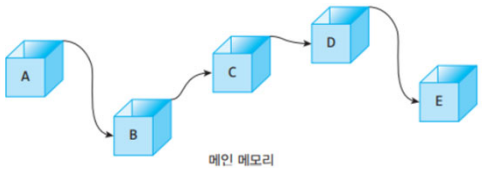
- 포인터와 동적 메모리 할당
  - 동적으로 할당된 메모리는 반드시 포인터에 저장
  - 그래야 사용할 수도 있고 해제할 수도 있다.

동적 메모리 할당과 해제

```
int* pi = (int*)malloc(sizeof(int)*10);
Polynomial* pa = (Polynomial*)malloc(sizeof(Polynomial));
pi[3] = 10;      // 동적 할당된 메모리를 배열처럼 사용
pa->degree = 5; // 동적 할당된 다항식 구조체의 멤버 변경
...
free(pi);
free(pa);
```

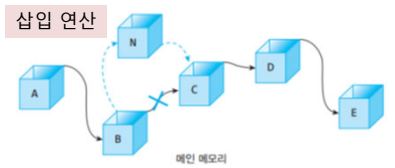
포인터의 응용 : 연결된 표현

- **Linked Representation ↔ 배열**
  - 항목들을 노드(node)라고 하는 곳에 분산하여 저장
  - 다음 항목을 가리키는 주소도 같이 저장
    - 노드 (node) : <항목, 주소> 쌍
  - 노드는 데이타 필드와 링크 필드로 구성
    - 데이타 필드 – 리스트의 원소, 즉 데이타 값을 저장하는 곳
    - 링크 필드 – 다른 노드의 주소값을 저장하는 장소 (포인터)
  - 메모리안에서의 노드의 물리적 순서가 리스트의 논리적 순서와 일치할 필요 없음

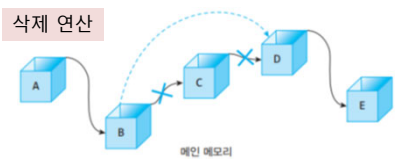


연결된 표현의 장단점

- 장점
  - 삽입, 삭제가 보다 용이하다.
  - 연속된 메모리 공간이 필요없다.
  - 크기 제한이 없다



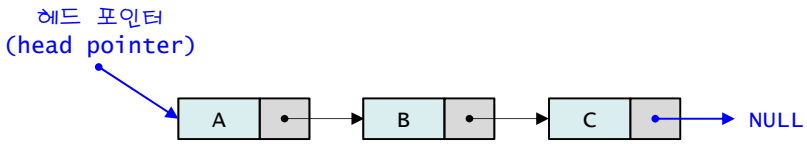
- 단점
  - 구현이 어렵다.
  - 오류가 발생하기 쉽다.



연결 리스트의 구조

- 노드

- 헤드 포인터(head pointer)
  - 리스트의 첫 번째 노드를 가리키는 변수

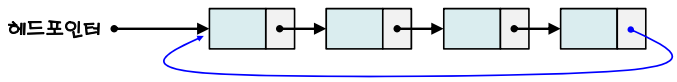


15

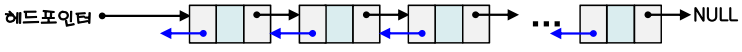
연결 리스트의 종류

- 단순 연결 리스트(singly linked list)

- 원형 연결 리스트(circular linked list)



- 이중 연결 리스트(doubly linked list)



16

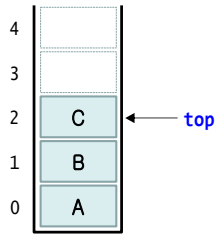
연결 리스트로 구현한 스택

- 노드 구조체

```
typedef int Element;
typedef struct LinkedNode {
        Element data;
        struct LinkedNode* link;
} Node;
```

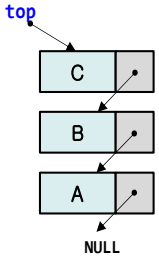
- 배열과 연결리스트를 이용한 스택의 비교

```
Element data[MAX_STACK_SIZE]
int top;
```



배열을 이용한 스택

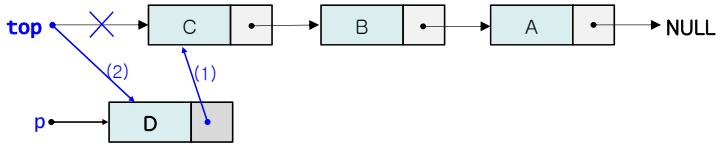
```
Node* top;
```



연결 리스트를 이용한 스택

삽입 연산

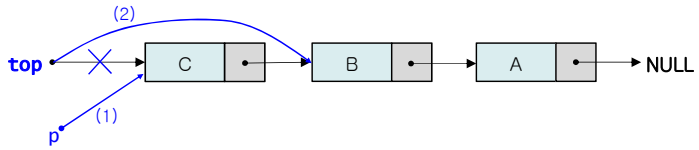
① 노드 D의 링크 필드가 노드 C를 가리키도록 한다: p->link = top;
② 헤더 포인터 top이 노드 D를 가리키도록 한다: top = p;




```
void push(Element e)
{
  Node* p = (Node*)malloc(sizeof(Node));
  p->data = e;
  p->link = top; // (1)
  top = p;       // (2)
}
```

삭제 연산

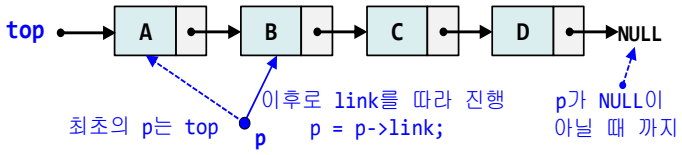
① 포인터 변수 p가 노드 C를 가리키도록 한다: p = top;
② 헤더 포인터 top이 B를 가리키도록 한다: top = p->next;
③ 마지막으로 변수 p의 값을 반환한다: return p;



삭제 연산

```
Element pop()
{
  Node* p;
  Element e;
  if (is_empty()) error("에러");

  p = top;       // (1)
  top = p->link; // (2)

  e = p->data;
  free(p);
  return e;
}
```

전체 노드 방문 연산



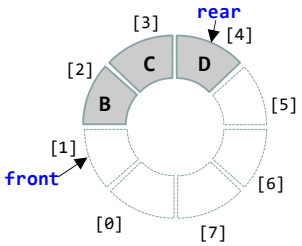

```
int size()
{
        Node *p;
        int count = 0;

        for (p = top; p != NULL; p = p->link)
                count++;

        return count;
}
```

21

연결리스트로 구현한 큐

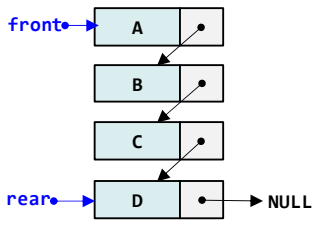
- 배열과 연결리스트를 이용한 큐의 비교

```
Element data[MAX_QUEUE_SIZE]
int front;
int rear;
```



배열을 이용한 원형 큐

```
Node* front;
Node* rear;
```

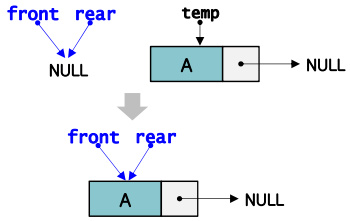


연결 리스트를 이용한 큐

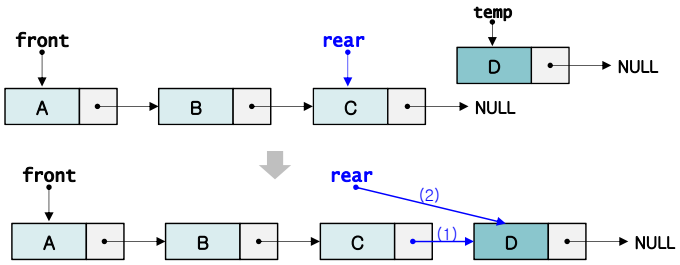
22

삽입 연산

- 공백상태의 삽입



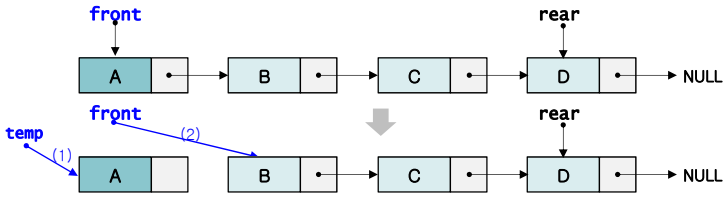
- 비 공백상태의 삽입



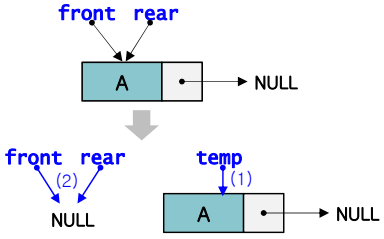
23

삭제 연산

- 노드가 두 개 이상



- 노드가 하나인 경우



24

► 연결리스트의 응용

연결리스트를 사용한 기억장소 관리

* 기억장소 관리 기법

(1) 정적 기억장소 관리(static storage management)
: 기억 장소의 크기를 고정적으로 분할한 후 그 크기에 맞는 job들을 저장.

(예)

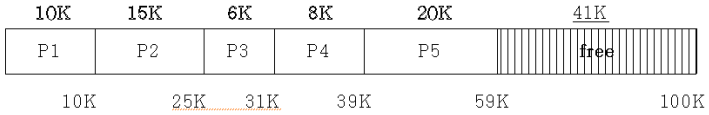
P1	P2	P3	P4	P5
10K	10K	20K	15K	30K

단점 : 분할된 기억 장소의 크기가 고정되어 있으므로 낭비가 심하고 multi programming에 부적합

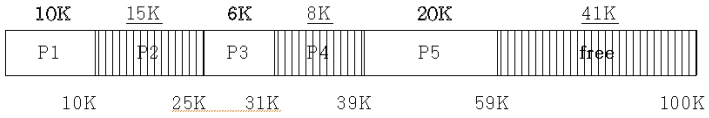
(2) 동적 기억장소 관리(dynamic storage management)
: multi programming을 실현하기 위하여 기억 장소의 크기를 가변적으로 분할한 후 그 크기에 맞는 job들을 저장.

(예)

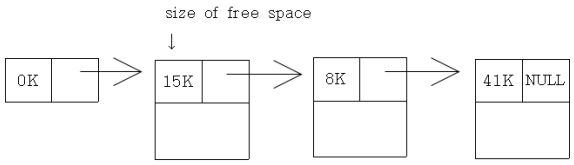
- 현재 상태



- P2와 P4의 실행 완료

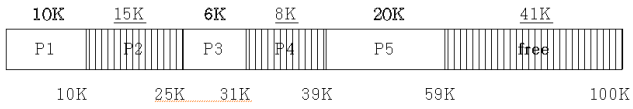


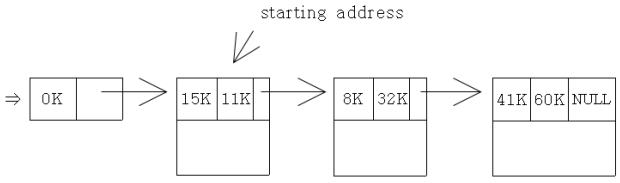
- 기억 장소의 상태에 대한 linked list 표현



27

- P2와 P4의 실행 완료





* 기억 장소 할당 알고리즘(memory allocation method)

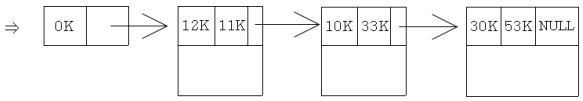
: 새로운 job을 기억 장소에 저장시키는 원칙.
first-fit, best-fit, worst-fit

28

① first-fit

: 가장 먼저 발견되는 free space 중 적당한 크기의 영역을 할당.

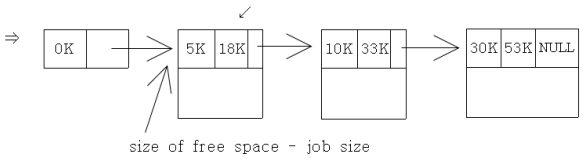
free space를 linked list로 표현하면



후속적으로 P5(7K), P6(5K), P7(9K), P8(22K)의 job이 들어온다면,

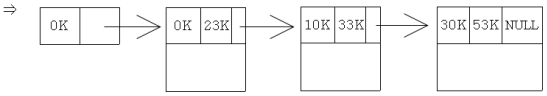
P5 저장

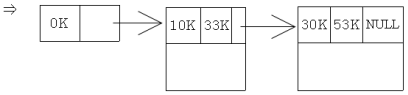
starting address + job size



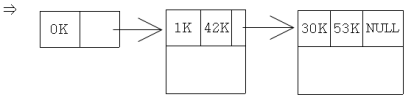
29

P6 저장



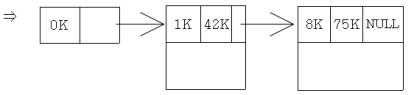


P7 저장



P8 저장

fragmentation



1) first-fit

- 장점 : 알고리즘이 간단
- 단점 : 주소가 큰 쪽에서 large free space가 몰리고 앞쪽은 fragmentation(단편화)이 많이 발생

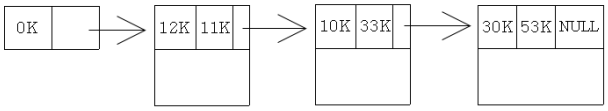
30

② best-fit

: job이 저장되기에 충분한 free space 중 크기가 가장 적합한 영역 할당

free space를 linked list로 표현하면

⇒

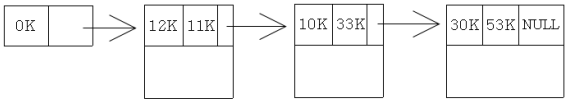


후속적으로 P5(7K), P6(5K), P7(9K), P8(22K)의 job이 들어온다면,

31

free space를 linked list로 표현하면

⇒

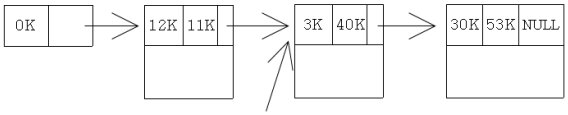


후속적으로 P5(7K), P6(5K), P7(9K), P8(22K)의 job이 들어온다면,

P5 저장

starting address + job size

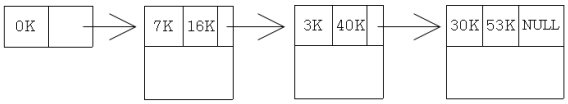
⇒



size of free space - job size

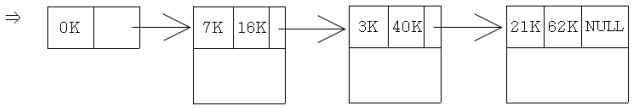
P6 저장

⇒



32

P7 저장



P8 저장 불가

- 장점 : 적당한 크기의 free space를 찾기 위해 large free space를 쪼개는 일이 드물다.
- 단점 : 작은 fragmentation 발생 --> 2) best-fit

③ worst-fit

: best fit의 반대 개념으로 큰 size에 우선배정

문제점) 보편성 결여, mid size 에서는 대체로 유용함

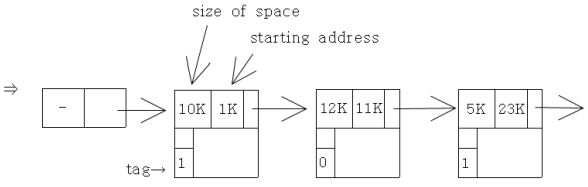
※ 실험적으로 first-fit이 양호한 것으로 증명!!!

33

*garbage collection

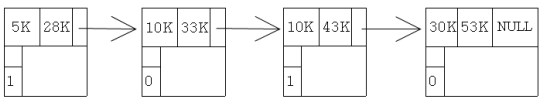
: 기억 장소 할당 알고리즘에 의해 발생된 fragmentation들을 모아서 하나의 큰 free space로 만들어주는 기법

(예)



tag0 : free space

tag1 : 사용중



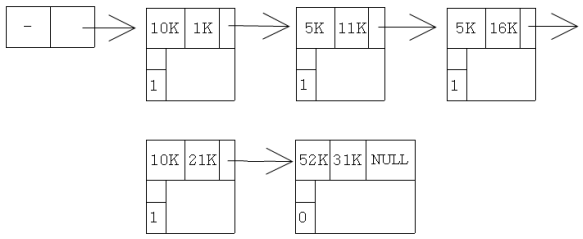
34

method

1) 각 node의 tag bit를 check하여
2) tag가 0일 때부터 시작하여 다음 node의 tag가 1인 경우, tag가 0인 node의 starting address는 변하지 않고 tag가 1인 node의 starting address는 tag가 0인 node의 starting address에 tag가 1인 node의 size를 더한다. 또한, tag가 1인 node의 size와 tag가 0인 node의 size를 바꾼다.
3) 만일 tag가 0인 node가 2개 연속적으로 있으면 뒤 node의 size와 앞 node의 size를 더해서 앞 node의 size를 update시킨 후 뒤 node는 삭제한다.

최종 상태

⇒



◆ 정리

- 포인터
  - *, &, ->, . 의 사용
  - 동적 메모리 할당
- 연결 리스트
  - 스택의 구현
  - 큐의 구현
- 리스트를 이용한 기억장소 관리
  - 기억장소 할당 알고리즘(first fit, best fit, worst fit)
  - Garbage Collection