자료구조와실습 중간고사 대비

컴퓨터공학전공 2020112736 안성현

2. C Programming

- #include <stdio.h> - Preprocessor(전처리기) - standard I/O function을 포함시킴.
- void main(void) - 모든 프로그램이 포함하고 있는 함수
- 변수 선언 - 모든 변수는 사용되기 전에 반드시 데이터 타입과 함께 선언되어야 함.

> 💡 Window X86 (32 bit) 기준 Data Type 별 크기
> - char : 1 byte
> - int : 4 byte
> - float : 4 byte
> - double : 8 byte

- getchar(), putchar() - 키보드 입력, 출력
- Conditional Operator(expr1? expr2 : expr3) - expr1을 평가하고 참이면 expr2, 거짓이면 expr3이 이 수식의 값

> 💡 Function Definition
> return type function_name(parameter list) {
> declarations
> statements
> }

return type, parameter list에 void가 있다면? : 각각 리턴하는 값 X, 인자를 갖지 않음.

> 💡 전치 : ++i; (Semicolon 전에 수행), 후치 : i++; (Semicolon 후에 수행)

!!!! 함수는 사용되기 전에 먼저 선언되어야 함

> 💡 Call by Value
> 함수 호출 시 매개변수는 그 값만이 전달됨. 변수가 함수로 전달되어도, 호출한 함수의 변수 값은 변경되지 않음.

💡 Array Initialization
* {}를 이용한 초기화 ex) int array[5] = {1, 2, 3, 4, 5};
* !!!! 명시적으로 초기화되지 않으면 시스템은 모든 원소를 0으로 초기화
* 문자열의 경우 char s[] = “abc”; → char s[] = {’a’, ‘b’, ‘c’, ‘\0’(null문자)};

💡 Pointer
프로그램의 메모리의 주소를 다루기 위해 사용
** 주소 연산자 &v : 변수 v값이 저장된 주소 (메모리 위치)
** 포인터 변수 int *p : 주소를 값으로 갖는 변수, p는 정수에 대한 포인터(address), *p는 p의 주소인 변수의 값을 나타냄.
ex) *p = 10(주소 p가 가리키는 값), p = 001A (p의 주소) (‘*’가 가리킨다라는 표시로 생각하면 될 듯)

```
// example
int main(void) {
  int i = 7, *p = &i;
  printf("%s%d\n%s%p\n", "Value of i : ", *p, "Location of i : ", p);
  return 0;
}

// Value of i : 7
// Location of i : effffb24
```

💡 Call by reference
하기위해서는 매개변수에 주소값을 전달
1. 매개변수를 포인터형으로 사용
2. 함수 body부분에서 *사용
3. 함수를 호출할 때 &(주소 연산자) 사용

(매개변수에 *, 호출할 때 & 사용하기.)

```
// example
void swap(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main() {
    int x = 3;
    int y = 1;
    swap(&x, &y);
    printf("%d %d\n", x, y);

    return 0;
}
```

- Pointer Arithmetic

배열의 이름 = 주소 = 포인터 값.

```
// example
int main() {
    int a[] = {1, 2, 3, 4};
    int *p;
    p = a;     // p = a[0]
    printf("%d\n", *p); // a[0]
    p++;
    printf("%d\n", *p); // a[1]
    printf("%d\n", *(p+1)); // a[2]
}
```

- 매개변수가 배열이면 실질적으로는 포인터로 보아야한다. (배열의 주소가 전달, 원소 자체의 복사 X)
- malloc() - 메모리 할당, 주소 리턴

```
int n = 1; // 정적 할당
int *p; // int 할당 X (동적 할당 필요)
*p = 5; // ERROR (why? : p가 주소 공간만 할당을 받은 상태라서)
p = malloc(sizeof(int));
*p = 1; // ok
```

- String은 char형의 1차원 배열 ex) {'h', 'e', 'l', 'l', 'o', '\0'}
- 'a' ≠ "a" - 전자는 char(1 byte), 후자는 + null문자 (2 byte)
- 문자열 상수는 배열의 이름과 같이 포인터로 취급
- 멤버 접근 연산자 (구조체) '.' : 일반 변수(*p), '→' : 포인터 변수(p)
- 구조체로 함수 이용해서 값 바꾸고 싶을 때, 1. 리턴 타입 구조체로 해서 구조체 리턴 2. Call by reference

- when Developing a Large Program : 헤더파일 분리
- 헤더파일에 들어가야 할거 #include, #define, 구조체, union, 함수의 기본형
- 이렇게 만든 헤더파일이 .c 파일들 상단에 include → 프로그램들을 하나로 묶는 역할

3. Basic Concepts (Algorithms and Complexity)

- System Life Cycle : 요구사항 → 분석 → 설계 → 코딩(구현) → 검증
- Algorithm의 기준 : 명백성, 유효성, 유한성
- Sorting, Searching Algorithm 숙지

💡 ADT(Abstract Data Type)
일반적으로 Data Type은 데이터 객체와 연산으로 이루어짐. ADT, 추상화된 데이터 타입은 그 뜻에 따라 데이터 객체와 연산이 무엇인지는 정의되지만, 데이터나 연산의 구현은 드러나지않음. 또한, ADT 연산에 대한 내부의 표현이나 구현설명은 없음. (구현에 독립적이다.)

예를 들어, ADT stack의 데이터 객체는 원소들의 리스트, 연산은 Push, Pop임. 하지만 Push, Pop이 어떻게 구현되있는지는 드러나지 않음.

- ADT 성능에 대한 평가

1. 원래의 기능대로 정확하게 작동하는지,
2. 문서화?
3. 함수 효과적 사용
4. 가독성
5. 메모리 공간 효율적 사용하는지
6. 실행시간

💡 공간 복잡도 (Space Complexity)
메모리 공간에 관한 복잡도
S(P) = c + SP(I)
코드에 대한 공간 + 실행에 필요한 공간

```
//example
float rsum(float list[], int n) {
if (n == 0) return 0;
return rsum(list[], n-1) + list[n - 1];

// size of n, size of address of list (함수에 배열 불러올 때 주소를 불러오니까), return address
// = 4 + 4 + 4
// S(n) = 12n bytes.
```

💡 시간 복잡도 (Time Complexity)
프로그램 실행 시간에 관한 복잡도
T(P) = c + TP(I)
컴파일 시간 + 실행 시간

```
//example
float sum(float list[], int n) {
float sum = 0;
int i;
count++; // 배정문 count
for (i = 0; i < n; i ++) {
count++; // for문 count
```

```
sum += list[i];
count++; // 배정문 count
}
count++; // for문의 마지막 실행 count
count++; // return count
return sum;
}

// T(n) = 2n + 3
```

- 복잡도 계산 - 정확한 시간은 의미 X. 입력 크기에 따른 시간 증가 정도만 분석

ex) A : 4n, B : 100n 증가함에 따라 우위가 달라짐. → 개략적으로만 계산을 해준다.

→ big O notation : 연산 횟수를 대략적(점근적)으로 표기하기 위해 사용

위의 T(n) = 2n + 3을 예시로 보면, 2n + 3 ≤ 3n, O(n) = 2n + 1

4. Arrays

- 배열 : index, value의 한 쌍의 집합, 메모리 위치 연속적
- ADT of Arrays (데이터 객체와 연산자에 대해)

데이터 객체 : <index, value>의 집합, index는 1차원 또는 다차원

연산자(함수) : Array Create(j, list) = j차원의 배열을 리턴

Item Retrieve(A, i) = index i값에 관련된 항목 리턴

Array Store(A, i, x) = 새로운 <i, x> 쌍을 A 배열에 삽입, 배열 A 리턴

- 1차원 배열에서의 메모리 주소

list[0] 주소 : a, list[1] 주소 : a + sizeof(int), . . .

주소를 포인터p로 받으면 다음 인덱스의 주소는 p+1

배열의 매개 변수 전달 방식 (in C) : Call by Value

- 구조체와 union : 타입이 다른 데이터들을 구조화시킨 데이터

멤버 연산자 : '.'(구조체로 선언된 구조체에 접근), '→'(포인터로 선언된 구조체 접근)

- 구조체 속에 또 다른 구조체 정의 가능

```
typedef struct {
  int year;
  int month;
  int date;
} when;

typedef struct {
  char name[];
  int num;
  when n1; // 미리 선언되어서 들어가야 함
} student_info;

// 선언할 때는 student_info s1;
// 접근할 때는 s1.n1.year = ~~;
```

- Union은 멤버들이 메모리 공간을 공용으로 사용.
- Self-Reference : 구조체 멤버가 구조체 자신을 가리키는 포인터가 있는 구조.

```
typedef struct {
  int data;
  listnode *link;
} listnode;

int main() {
  listnode item1, item2;
  item1.data = 10;
  item1.link = &item2; // item1.link 포인터 변수가 item2의 위치.
  item1.link -> data = 20; // item2의 data (list1.link가 item2의 포인터이니까 '->' 사용)
}
```

- Polynomial ADT (다항식의 ADT)

$$A(x) = 3x^9 + 4x^2 + 10x + 20$$

Polynomial ADT의 자료구조 : coef(배수부분)와 exp(지수부분)를 멤버로 갖는 구조체의 배열

A(x)에 대한 배열	coef	exp
list[0]	3	9
list[1]	4	2
list[2]	10	1
list[3]	20	0

▼ Polynomial ADT의 연산자 (함수)

Coefficient Coef(poly, exp) : 계수 리턴

Polynomial Attach(poly, coef, exp) : <coef, exp> 가지는 poly 삽입 후 리턴

Polynomial Remove(poly, exp) : 지수가 exp인 항 삭제후 리턴

Polynomial SingleMult(poly, coef, exp) : 다항식 리턴

Polynomial Add(poly1, poly2) : 다항식의 합

Polynomial Mult(poly1, poly2) : 다항식의 곱

- Sparse Matrix (희소 행렬)

행렬의 원소가 거의 0일 때, 일반 행렬은 메모리 공간 낭비

→ Solution : 0이 아닌 원소 <행, 열, 원소>로 저장.

$$\begin{pmatrix} 0 & 0 & 0 & 4 \\ 0 & 5 & 0 & 0 \end{pmatrix} -> list < 2,4,2 >, < 0,3,4 >, < 1,1,5 >$$

Sparse Matrix의 자료구조 : 행, 열, 원소를 멤버로 가지는 구조체의 1차원 배열, 원소 개수가 배열의 크기.

```
typedef struct {
  int row;
  int column;
  int value;
} sparse_matrix;
```

ex) int a[1000][1000] (일반 2차원 행렬) vs sparse_matrix a[200] (희소 행렬) → 메모리 공간 절약 !

a[0].row = 전체 행의 개수, a[0].column = 전체 열의 개수, a[0].val = 전체 원소의 개수

Matrix Transpose의 Time Complexity - O(row*col) vs O(col + element)

▼ Sparse Matrix 의 함수

sparse_matrix Create(max_row, max_col) : 행렬 생성 리턴

Sparse_matrix Transpose(a) : a의 전치행렬 리턴

Sparse_matrix Add(a, b) : 차원이 같으면 합 리턴

Sparse_matrix Multiply(a, b) : a의 column = b의 row → (a_row) by (b_column) 행렬 리턴

- Multi-Dimention Array

행 우선 vs 열 우선 → 메모리에 행 먼저 저장하냐, 열 먼저 저장하냐 인듯???

주소 계산 → base + ~~

5. Stacks And Queues

- Stack ADT

Last-In-First-Out

top은 -1에서 시작

Push에선 top++

Pop에선 —top

▼ Stack ADT의 함수

Stack CreateS(max_size) : max_size 크기의 스택 생성, 리턴

Void Push(int item) : IsFull() 확인하고 ++top : item

Element Pop() : IsEmpty() 확인하고 top—

Boolean IsEmpty() : top < 0

Boolean IsFull() : top >= max_size - 1

- Queue ADT

First-In-First-Out

first, rear는 -1에서 시작

AddQ에선 ++rear

DeleteQ에선 ++front

Queue가 Full이 되면 front, rear 값을 모두 조정해주어야 한다는 단점 존재

▼ Queue ADT의 함수

Queue CreateQ(max_size) : max_size 크기의 큐 생성, 리턴

void AddQ(int item) : IsFullQ() 확인하고 ++rear : item

void DeleteQ() : IsEmptyQ() 확인하고 ++front

Boolean IsEmptyQ() : rear = front (원소 없음)

Boolean IsFull() : rear >= max_size - 1

- Circular Queue

front : 첫 번째 원소의 하나 앞, rear : 마지막 원소

modular 연산을 사용해 삽입과 삭제

Full : (rear + 1) % max_size = front

Empty : rear = front

- 수식의 표현

중위 표기 : 우리가 사용하는 표기법

후위 표기 : 연산자가 피연산자 후위에 위치

전위 표기 : 연산자가 피연산자 전위에 위치

- 후위 표기법

괄호 사용 X, 연산자 우선순위 X (왼쪽에서 오른쪽으로 그냥 계산)

- 후위 표기 연산 방법

1. 연산자 나오기 전까지 피연산자 스택에 저장
2. 연산자가 나오면 연산에 필요한 만큼 피연산자 스택에서 가져와서 연산하고 결과 스택에 저장

→ 연산자 나오기전까지 피연산자 Push(), 연산자 나오면 필요한 피연산자 Pop()해서 반환, 연산하고 결과 Push(). 마지막에는 Pop() 해서 결과 값 반환

6. Linked Lists

- Pointers

배열의 단점 (정적 할당으로 인한 크기 고정, 원소 삽입삭제 번거로움)

→ Solution : Pointer(Link) 사용 → 동적으로 공간 생성, 원소 삽입삭제 효율적

int *p로 선언

‘*’ : 지시 연산자, ‘&’ : 주소 연산자로 객체 사용

- malloc() : 동적으로 기억장소 할당
- free() : 동적 할당 해제

```
int i, *p;
p = (int *)malloc(sizeof(int)); // 정수형 사이즈의 기억공간 동적 할당
*p = 10;
free(p); // 동적으로 할당 받은 기억공간 해제
```

- Linked List (연결 리스트)

화살표로 표시된 링크를 가진 노드들의 순열

크기 미리 안정해져있음

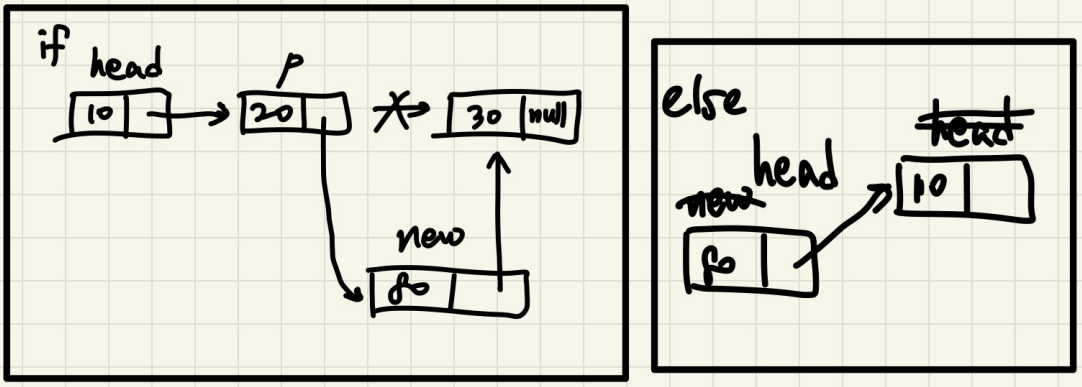
리스트의 구성 : data, link

data : 실제 자료

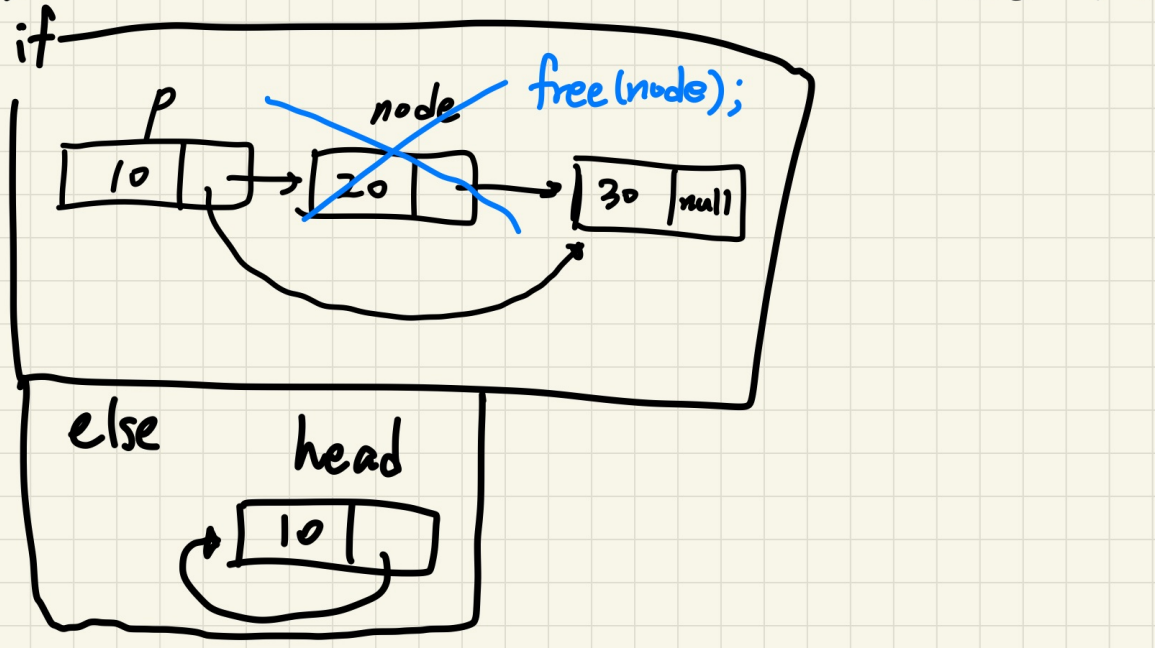
link : 다음 노드 주소

```
#include <stdio.h>

typedef struct list_node { // list_node를 먼저 선언해야 link_node* 를 선언할 수 있음.
    int data;
    list_node *link; // 이 구조체의 포인터
} list_node;

int main() {
    list_node *a;
    a = (list_node*)malloc(sizeof(list_node)); // 구조체 포인터 형태로 list_node 사이즈 만큼의 기억 공간을 동적으로 할당
}
```

```
void Insert(list_node * p, list_node* new) {
    if(p) {
        new->link = p->link;
        p->link = new;
    }
    else{  ~> p가 없으면 맨 앞에 삽입
        new->link = head;
        head = new;
    }
}
```



```
void delete(list_node* p, list_node* node) {
    if(p) {
        p->link = node->link (p->link->link)
    }
    else{  → p가 없으면 맨 앞의 노드 삭제
        head = head->link;
    }
    free(node); → 삭제된 노드는 동적 할당 해제
}
```



```
void Print_list(list_node* head) {
    list_node* p = head;
    while (p != NULL) {
        printf("%d ", p->data);
        p = p->link
    }
}
void Search_list(list_node* head. char* word){
    list_node* p = head;
    while(p) {
        if(!strcmp(p->data.word)) {
            return p;
        }
        p = p->link;
    }
    return NULL; ~ 검색결과 없음.
}
```

▼ Linked Stack

stack* top

- void push(element item) : 스택을 top 앞으로 쌓는다고 생각.

  temp = (stack*)malloc(sizeof(stack))

temp → item = item

temp → link = top

top = temp (Stack 특성 생각 !!)

- element pop() : pop한 item 리턴해주는 특성 생각하기.

  리턴, 할당 해제 용도로 element item, stack* temp 선언

  item = top → item (리턴해주기위해서 잠시 보관)

  temp = top (할당 해제 해주기 위해서 잠시 보관)

  top = top → link

  free(temp) 보관해뒀던 top(temp) 할당 해제

  return item

▼ Linked Queue

queue* front, rear

- addQ(element item) : 앞에다가 추가

  temp = (queue*)malloc(sizeof(queue))

  temp → item = item

  temp → link = NULL

  rear → link = temp;

  rear = temp

- element delete() : 뒤에서 부터 삭제

  리턴, 할당 해제 용도로 element item, queue* temp 선언

  item = front → item

  temp = front

  front = front → link

  free(temp)

  return item

▼ Linked Polynomial

coef, exp로 데이터가 이루어진 노드

padd(a, b) : 두 다항식 합 구하는 함수.

첫 a, b 노드 (head) 정해주고 두 다항식 지수부분 비교해서 처리해주는 방식

그림으로 생각해보기

▼ Doubly Linked List

이중 연결 리스트 : 양방향으로 연결 (llink, rlink)

연결해줄 때는 → → 이중으로 사용하는 곳도 있음.

insert, delete

- 연산들의 Time Complexity
  - 비정렬 리스트

	Find	Insert	Delete
Array	O(n)	O(1)	O(n)
Linked List	O(n)	O(1)	O(n)

  - 정렬 리스트

	Find	Insert	Delete
Array	O(log n) binary	O(n) (log n + n)	O(n) (log n + n)
Linked List	O(n)	O(n) (n + 1)	O(n) (n + 1)