Pintos Project 0-2 : Pintos Data Structure

과목 명 : [CSE4070] 운영체제 1반

담당 교수 : 박성용 교 수 님

학번 : 20160393

이름 : 서인호

프로젝트 제목 : pintos project 0-2 Pintos Data Structure

제출일 : 2019년 10월 6일

**I. List**

0. Struct list\_item

Struct list\_elem\* 와 int data를 멤버 변수로 갖는 구조체이다. list\_elem 은 리스트의 head와 tail 사이에 존재하고, data는 해당 주소값이 갖는 데이터를 의미한다.

1. void list\_init(struct list\*, char\*)

Parameter: 새롭게 생성할 리스트의 주소값과 리스트의 이름으로 사용할 문자열 주소

return: 없음

function: 파라미터로 전달받은 리스트 자료형 주소에 빈 리스트를 생성한다. Head와 tail을 설정하고, 리스트를 구분하기 위해 이름을 리스트 구조체 안에 저장한다. 이를 위해 기존의 struct list 의 구조체 안에 ‘char name[20]’의 영역을 추가하였다. 그리고 전역 변수로 struct list 자료구조를 10개 저장할 수 있는 배열 ds\_list을 선언하여 각 리스트를 구분하였따. 또한, 정적 할당한 배열 크기 10개 만큼의 리스트가 이미 만들어져 있다면 추가로 리스트를 만들지 않도록 현재 만들어진 리스트 개수를 체크하는 list\_cnt 변수를 사용해 리스트 개수를 관리했다.

2. int check\_list(char\*)

Parameter: 찾고자 하는 리스트 이름을 문자열 포인터로 입력받는다.

return: 전역변수로 선언된 ds\_list 배열 중에 파라미터로 전달받은 문자열과 같은 이름을 가진 자료구조가 있는지 확인한다. 만약 일치하는 자료구조가 존재하면 해당 자료구조의 인덱스 값을 리턴한다. 해당 이름의 자료구조가 존재하지 않으면 -1을 리턴한다.

Function: 입력받은 리스트의 이름이 이전에 create 명령어로 생성된 리스트 중에 있는지 확인한다.

3. delete

Function : 별도의 함수를 구현하지는 않았으나, 메인 함수에서 delete <list\_name> 의 형태로 입력이 들어오면 메인 함수에서 check\_list 함수를 수행하여 입력된 이름의 리스트가 있는지 확인한다. 해당 이름의 리스트가 존재하면 배열에서 해당 리스트의 인덱스+1의 리스트를 지워질 리스트의 위치로 옮긴다. 이 작업을 리스트 삭제 전에 있던 리스트의 개수 – 지워질 리스트의 인덱스 수만큼 반복한다. 그리고 마지막에 사용하던 리스트 인덱스는 초기화한다.

4. dumpdata

Function: 별도의 함수 구현 없이 사용자로부터 dumpdata <list\_name> 입력을 받으면 check\_list 함수를 사용하여 사용자가 입력한 이름의 리스트가 있는지 확인한다. 해당 리스트가 존재하면 list\_entry 메크로를 이용해 struct list\_item 포인터를 얻어 list\_item의 data 멤버에 접근해 리스트 안에 저장된 데이터 값을 프린트한다.

5. void list\_push\_front(struct list\*, struct list\_elem\*)

parameter: 작업을 수행할 리스트의 주소값과 새로 추가할 list\_elem 주소값을 받는다.

return: 없음

function: 메인 함수에서 사용자로부터 list\_push\_front <list\_name> <data>를 입력받으면 check\_list 함수를 사용해 사용자가 입력한 이름의 리스트가 존재하는지 확인한다. 해당 리스트를 찾으면 struct list\_item 구조체를 새롭게 생성하고, new->data 에 사용자가 입력한 값을 저장한다. 그리고 list\_push\_front 함수를 호출하는데, 함수의 첫 번째 파라미터로 전달받은 리스트 맨 첫번째, 즉 head->next에 두 번째 파라미터로 전달받은 list\_elem의 주소값을 삽입한다. 이때 list\_insert 함수가 내부적으로 사용된다.

6. void list\_push\_back(struct list\* struct list\_elem\*)

parameter: 작업할 리스트의 주소값과 추가할 list\_elem 의 주소값을 받는다.

return: 없음

function: 메인 함수에서 사용자로부터 list\_push\_back <list\_name> <data>를 입력받으면 check\_list 함수를 사용해 사용자가 입력한 이름의 리스트가 존재하는지 확인한다. 해당 리스트를 찾으면 struct list\_item 구조체를 새롭게 생성하고, new->data 에 사용자가 입력한 값을 저장한다. 그리고 list\_push\_back 함수를 호출하는데, 함수의 첫 번째 인자로 입력받은 리스트의 맨 마지막, 즉 tail->prev에 두 번째 인자로 받은 list\_elem의 주소값을 추가한다. 이 작업을 위해 함수 내부에서 list\_insert 함수가 추가로 사용된다.

7. struct list\_elem\* list\_pop\_front(struct list\*)

parameter: 함수에서 작업할 리스트의 주소값을 받는다.

return: 함수 수행 전 파라미터로 전달받은 리스트의 첫번째 주소, 즉 head->next 의 값을 반환한다.

Function: 프로그램에서 사용자가 list\_pop\_front <list\_name> 을 입력하면 check\_list 함수를 사용해 사용자가 입력한 이름의 리스트가 존재하는지 확인한다. 해당 리스트를 찾으면 list\_pop\_front 함수를 호출한다. List\_pop\_front 함수는 파라미터로 받은 리스트의 첫번째 주소, 즉 head->next을 삭제한다. 이 작업을 수행하기 위해 함수 내에서 list\_front 함수와 list\_remove 함수를 추가로 사용한다.

8. struct list\_elem\* list\_pop\_back(struct list\*)

parameter: 함수에서 작업할 리스트의 주소값을 받는다.

return: 함수 수행 전 파라미터로 전달받은 리스트의 마지막 주소, 즉 tail->prev 의 값을 반환한다.

Function: 프로그램에서 사용자가 list\_pop\_back <list\_name> 을 입력하면 check\_list 함수를 사용해 사용자가 입력한 이름의 리스트가 존재하는지 확인한다. 해당 리스트를 찾으면 list\_pop\_back 함수를 호출한다. List\_pop\_front 함수는 파라미터로 받은 리스트의 마지막 주소, 즉 tail->prev을 삭제한다. 이 작업을 수행하기 위해 함수 내에서 list\_back 함수와 list\_remove 함수를 추가로 사용한다.

9. struct list\_elem\* list\_front(struct list\*)

parameter: 함수에서 작업을 수행할 리스트의 주소값을 받는다.

return: 파라미터로 받은 리스트의 첫번째 list\_elem, 즉 head->next의 주소값을 반환한다.

Function: 사용자가 list\_front <list\_name> 의 명령어를 입력했을 때 사용자가 입력한 이름의 리스트에 저장된 첫번째 list\_elem 주소값을 얻는다. 그리고 list\_entry 메크로를 이용해 반환된 주소값의 list\_item 구조체를 얻고, 이렇게 얻은 list\_item의 data 멤버에 있는 값을 화면에 출력한다. list\_push\_front와 list\_pop\_front 와 같은 함수 구현을 위해 사용되기도 한다.

10. struct list\_elem\* list\_back(struct list\*)

parameter: 함수에서 작업을 수행할 리스트의 주소값을 받는다.

return: 파라미터로 받은 리스트의 마지막 list\_elem, 즉 tail->prev의 주소값을 반환한다.

Function: 사용자가 list\_back <list\_name> 의 명령어를 입력했을 때 사용자가 입력한 이름의 리스트에 저장된 첫번째 list\_elem 주소값을 얻는다. 그리고 list\_entry 메크로를 이용해 반환된 주소값의 list\_item 구조체를 얻고, 이렇게 얻은 list\_item의 data 멤버에 있는 값을 화면에 출력한다. list\_push\_back, llist\_pop\_back과 같은 함수의 구현을 위해 사용되기도 한다.

11. void list\_insert(struct list\_elem\*, struct list\_elem\*)

parameter: 첫 번째 인자로 새로운 element가 추가될 위치의 주소값, 두 번째 인자로 추가될 element의 주소값을 받는다.

return: 없음

function: 첫 번째 인자로 받은 element의 위치에 두 번째의 element를 추가한다. 즉, 함수가 수행된 이후에는 두 번째 element가 먼저 오고 두 번째 element는 원래의 위치보다 한 칸 뒤에 위치하게 된다. 파라미터의 첫 번째 인자를 a, 두 번째 인자를 b라고 하면, b의 next가 a가 되고, a의 prev가 b가 된다. 연결 관계를 유지하기 위해 a의 prev는 b의 prev가 되고, a의 prev의 next는 a가 아니라 b가 된다. 메인 함수에서는 사용자가 list\_insert <list\_name> <location> <data>를 입력하면 check\_list 함수로 해당 이름의 리스트를 찾고, list\_item의 자료형을 새롭게 할당해서 사용자가 입력한 data를 저장한다. 그리고 새로운 list\_elem 자료형을 현재 작업중인 리스트의 첫번째 element로 초기화한 후, 사용자가 입력한 location 수 만큼 이동시킨다. 이후 list\_insert 함수의 첫 번째 인자로 새로운 list\_elem과 두 번째 인자로 list\_item의 list\_elem 주소값을 넘긴다.

12. void list\_insert\_ordered(struct list\*, struct list\_elem\*, list\_less\_func \*, void\*)

parameter: 첫 번째 인자로 작업할 리스트의 주소값, 두 번째 인자로 추가할 element, 세 번째로 list\_less\_func로 정의된 정렬 함수, 네 번째로 NULL 값을 받는다.

return: 없음

function: 첫 번째 인자로 들어온 리스트의 element 값을 조사한 후에, 두 번째 인자로 들어온 element가 삽입될 위치를 결정해 삽입한다. 이때, element가 삽입될 위치는 세 번째 인자인 list\_less\_func에 오는 함수의 기능에 따른다. 현재 본 프로젝트에서는 ‘compare’라는 함수를 정의하여 list\_less\_func 에 전달하고 있는데, compare는 파라미터로 들어온 두 개의 struct list\_elem을 list\_entry 메크로를 통해 list\_item의 데이터 값을 비교한다. 그렇게 얻은 두 개의 값 중 첫 번째 인자 element 주소의 데이터 값이 두 번째 인자 element 주소의 데이터 값보다 작은 true를, 두 값이 같거나 첫 번째 데이터가 크면 false를 반환한다. 즉, 여기서 새로운 element가 삽입될 위치는 list\_less\_func 자료형인 compare 함수가 첫 번째 값이 두 번째보다 작을 때 참을 반환하도록 설계되었으므로, 오름차순에 따라 새로운 element가 삽입되게 된다. 메인 함수에서는 사용자가 list\_insert\_ordered <list\_name> <data>를 입력했을 때, check\_list 함수를 이용해 해당 리스트를 찾고, 새로운 list\_item 자료형인 new에 사용자가 입력한 data를 저장한다. 이후 new의 list\_elem 멤버에 저장된 element 주소값을 list\_insert\_ordered 함수의 두 번째 인자로 넘겨 사용자가 명시한 리스트에 새로 만든 데이터를 추가한다.

13. struct list\_elem\* list\_remove(struct list\_elem\*)

parameter: 삭제할 element의 주소값을 받는다.

return: 삭제된 element의 next, 즉 파라미터로 들어온 element의 다음 element를 반환한다.

Function: 파라미터로 전달받은 list의 element를 리스트 안에서 삭제한다. 이를 위해 파라미터로 들어온 element의 이전의 다음 항목인 list\_elem->prev->next를 삭제할 element의 다음 멤버로 연결하고 next에 대해서도 마찬가지 작업을 수행한다. 즉, 리스트 내부의 연결관계 상에서 해당 element의 연결고리를 끊어내는 것이다. 그리고 끊어진 element의 next를 반환한다. 여기서 파라미터로 들어온 element는 연결 관계에서 끊어졌기 때문에 일반적인 방법으로 더 이상 접근할 수 없게 되었을 뿐, 메모리 상 어딘가에 계속 존재한다. 메인 함수에서는 사용자가 list\_remove <list\_name> <index> 를 입력하면 check\_list 함수로 해당 리스트를 찾은 이후, list\_elem 자료형의 포인터로 사용자가 지정한 index 수 만큼 해당 리스트 내부를 연결고리를 따라 이동하여 연결고리를 제거하는 list\_remove 함수를 실행한다.

14. void list\_splice(struct list\_elem\*, struct list\_elem\*, struct list\_elem\*)

parameter: 첫 번째 인자로 element를 옮기고 싶은 위치, 두 번째와 세 번째 인자로 옮기려는 element의 범위를 지정한다.

return: 없음

function: 첫 번째 인자로 들어온 element의 앞으로 두 번째 element부터 세 번째 element까지 존재하는 연결관계의 모든 element를 옮긴다. 함수가 실행되기 전 두 번째 파라미터 세 번째 파라미터까지의 범위에 있던 element들은 원래 위치에서 삭제된다. 메인 함수에서 사용자가 list\_splice <list\_name> <index> <list\_name> <start\_index> <end\_index>를 입력하면 두 부분으로 나누어 사용자 입력을 처리한다. 첫 번째 부분은 list\_splice <list\_name> <index> 까지로, check\_list로 사용자가 입력한 이름의 리스트를 검색하고 list\_elem 포인터인 ptr로 사용자가 지정한 index 위치를 찾는다. 이 작업이 정상적으로 종료되면 두 번째 <list\_name> <start\_index> <end\_index>를 처리한다. 마찬가지로 check\_list로 사용자가 입력한 리스트가 유효한지 확인하고, list\_elem 자료형인 ptr2와 ptr3으로 start\_index와 end\_index 범위를 잡는다. 그리고 list\_splice 함수에 첫 번째 단계에서 구해 놓은 ptr과 두 번째 단계에서 구한 ptr2, ptr3을 파라미터로 넘겨 작업을 수행한다.

15. void list\_unique(struct list\*, struct list\*, list\_less\_func\*, void\*)

parameter: 작업할 두 개의 리스트와 작업을 수행할 기준이 되는 list\_less\_func 함수, 그리고 NULL 포인터를 받는다.

return: 없음

function: 첫 번째 리스트에 저장된 값 중에 중복된 값을 모두 두 번째 리스트에 옮겨 저장한다. 함수가 실행되고 나면 첫 번째 리스트에는 각각의 값이 단 한 개만 남게 된다. 리스트가 두 개가 아니라 한 개만 인자로 들어오면 옮겨지는 작업 없이 그냥 중복된 값이 삭제된다. 메인 프로그램에서 본 작업을 수행하기 위해 ‘unique’라는 함수를 별도로 정의했다. 사용자가 list\_unique 뒤에 입력하는 내용을 한 줄로 받고 스페이스를 기준으로 나눈다. 스페이스로 나눠지면 두 개의 입력이 들어왔다는 의미로 두 개의 리스트를 파라미터로 넘기고, 스페이스를 기준으로 잘라지지 않으면 하나만 입력된 것으로 판단하여 한 개의 리스트만 넘기고 두 번째 리스트는 NULL 포인터를 넘긴다. List\_less\_func 함수는 이전에 만든 compare 함수로 사용하였다.

16. bool list\_empty(struct list\*)

parameter: 비어있는지 확인하고자 하는 리스트의 주소값을 인자로 넘긴다.

return: 주어진 리스트가 비어있으면 true를 반환하고, 비어있지 않으면 false를 반환한다.

function: 주어진 리스트의 내용이 head->next가 tail이면 빈 리스트로 간주한다.

17. size\_t list\_size(struct list\*)

parameter: 크기를 알고자 하는 리스트의 주소값을 인자로 넘긴다.

return: 해당 리스트의 크기를 size\_t 자료형으로 반환한다.

Function: 주어진 리스트의 시작부터 끝까지 몇 개의 element가 있는지 조사하여 그 수를 얻는다. 메인 함수에서 이 함수를 실행한 결과를 프린트할 때에는 %zu를 사용한다.

18. struct list\_elem\* list\_max(struct list\*, list\_less\_func\*, void\*)

parameter: 가장 큰 element를 얻고싶은 리스트의 주소값, 오름차순 정렬 기준 함수, null 포인터를 인자로 넘긴다.

return: 해당 리스트 element 중 가장 큰 값의 element를 반환한다.

Function: 주어진 리스트의 내부를 돌며 가장 큰 값을 찾는다. 오름차순 정렬 함수로 compare 함수를 사용하였따.

19. struct list\_elem\* list\_min(struct list\*, less\_list\_func\*, void\*)

parameter: 가장 작은 element를 얻고싶은 리스트의 주소값, 오름차순 정렬 기준 함수, null 포인터를 인자로 넘긴다.

return: 해당 리스트 element 중 가장 작은 값의 element를 반환한다.

Function: 주어진 리스트의 내부를 돌며 가장 작은 값을 찾는다. 오름차순 정렬 함수로 compare 함수를 사용하였따.

20. void list\_sort(struct list\*, list\_less\_func\*, void\*)

parameter: 정렬하고자 하는 리스트의 주소값, 오름차순 기준함수, NULL 포인터를 인자로 넘긴다.

return: 없음

function: 주어진 리스트를 기준함수의 기능에 따라 정렬한다. 오름차순 함수인 compare를 사용했기 때문에 여기서는 주어진 리스트 element를 오름차순으로 정렬한다.

21. void list\_reverse(struct list\*)

parameter: 역순으로 정렬하고자 하는 리스트의 주소값을 인자로 넘긴다.

return: 없음

function: 파라미터로 주어진 리스트의 element를 거꾸로 정렬한다. 즉, list\_back이 list\_front가 된다.

22. void list\_swap(struct list\_elem\*, struct list\_elem\*)

parameter: 서로의 위치를 교환할 element 주소값 두 개를 인자로 넘긴다.

return: 없음

Function: 주어진 두 개의 element 위치를 바꾼다. 두 개의 위치가 변경됨에 따라 기존에 두 개의 element가 가지고 있던 연결관계도 모두 변경된다. 여기서는 a->next->prev, a->next, a->prev->next, a->prev의 순서로 연결관계를 수정하였다. 사용자는 작업하고자 하는 리스트의 이름과 인덱스 두 개를 입력한다. 메인 함수는 사용자의 입력 중 유효한 리스트를 찾아 두 개의 인덱스에 해당하는 element를 ptr과 ptr2에 저장하고, 이를 list\_swap 파라미터로 활용한다.

23. void list\_shuffle(struct list\*)

parameter: 내부 element의 순서를 변경할 리스트 주소를 인자로 넘긴다.

return: 없음

function: 주어진 리스트의 순서를 무자귀로 섞는다. 이 구현을 위해 랜덤 함수를 사용하였다. 우선 list\_size 함수를 통해 주어진 리스트의 크기를 구한 후, 리스트의 크기만큼 랜덤하게 두 개의 인덱스를 선택한다. 무자귀로 선택된 두 개의 인덱스에 대해 list\_swap 함수를 수행하였다. 즉, 주어진 리스트의 element 수 만큼 무자귀로 선택된 두 개의 element에 대해 list\_swap을 수행하였다.

**II. Hash Table**

0. struct hash\_item

Hash\_elem과 data 값을 멤버로 갖는 구조체를 선언하였다.

struct hash 구조체에 name이라는 멤버를 추가로 생성하여 각 hash 자료형을 구분할 수 있도록 하였다. 또한, 메인 함수에서 hash 자료형을 10 크기의 배열로 할당하고, hash\_cnt라는 변수를 두어 hash의 수를 관리하였따.

1. bool hash\_init(struct hash\*, char\*, hash\_hash\_func\*, hash\_less\_func\*, void\*)

parameter: 생성하고자 하는 hash의 주소값, hash 이름, hash 크기, hash 오름차순 기준정렬, NULL 포인터를 인자로 넘긴다.

return: 성공적으로 hash가 생성되면 true를, 메모리 부족 등의 원인으로 hash table이 생성되지 않았으면 false를 반환한다.

Function: 새로운 hash table을 생성한다.

2. delete

별도의 함수를 정의하지 않고, 사용자로부터 delete <hash\_name> 의 입력이 들어왔을 때 check\_hash 함수를 이용해 지우고자 하는 hash를 찾아 hash\_clear을 이용해 hash를 비우고 ds\_hash 배열에서 삭제한다. 이때, check\_hash는 check\_list의 기능과 같다. 다만, delete <hash\_name> 과 같은 입력 규칙 상 delete 뒤에 오는 자료형 이름이 어떤 자료형인지 알 수 없어 check\_list를 먼저 수행하고, 그 결과가 -1이면 check\_hash를 수행한다.

3. dumpdata

별도의 함수 정의 없이 사용자로부터 dumpdata <hash\_name> 의 입력이 들어오면 check\_hash를 이용해 해당 hash를 찾아 해당 hash의 값을 화면에 출력한다. 여기서 hash\_first 함수를 이용해 hash\_elem 포인터에 hash의 첫 번째 위치를 얻고, hash\_next 함수를 이용해 다음 인덱스로 넘기며 hash\_entry 메크로를 사용해 hash\_item의 값을 얻었다.

4. struct hash\_elem\* hash\_insert(struct hash\*, struct hash\_elem\*)

parameter: 작업하고자 하는 hash의 주소와 해당 hash에 추가하고자 하는 element 주소값을 인자로 넘긴다.

return: 새로 추가하고자 하는 데이터가 이미 hash table에 있는 경우에는 해당 element를 추가하지 않고 NULL 포인터를 반환한다. 만약 추가하고자 하는 element의 데이터 값이 현재 hash table에 존재하지 않으면, 해당 element를 추가한 후 추가된 element를 반환한다.

Function: 주어진 hash table에 새로운 hash element를 추가한다. 사용자가 hash\_insert <hash\_name> <data> 를 입력하면 메인 함수는 hash\_item 자료형을 새롭게 생성하여 사용자가 입력한 값을 hash\_item에 저장한 후, hash\_item의 hash\_elem 멤버를 hash\_insert 함수의 파라미터로 넘겨 작업을 수행한다.

5. struct hash\_elem\* hash\_replace(struct hash\*, struct hash\_elem\*)

parameter: 작업하고자 하는 hash의 주소값과 추가하고자 하는 element의 주소값을 인자로 넘긴다.

return: 새로 추가하고자 하는 값이 이미 hash table에 존재하는 경우, 기존의 값을 반환한다.

Function: 기본적으로 hash\_insert와 기능이 유사하다. 하지만 추가하고자 하는 값이 기존 hash에 존재하면 insert를 하지 않는 hash\_insert와 달리, hash\_replace는 기존에 저장되어있는 값과 무관하게 insert를 수행한다. 메인 함수에서 구현 방법은 hash\_insert와 같다.

6. void square(struct hash\_elem\*, void\*)

parameter: 값을 제곱하고자 하는 element의 주소값과 NULL 포인터를 인자로 넘긴다.

return: 없음

function: hash\_entry 메크로를 이용하여 주어진 element에 mapping 된 data를 얻어 그 값을 제곱한다.

7. void triple(struct hash\_elem\*, void\*)

parameter: 값을 세제곱하고자 하는 element의 주소값과 NULL 포인터를 인자로 넘긴다.

return: 없음

function: hash\_entry 메크로를 이용하여 주어진 element에 mapping 된 data를 얻어 그 값을 세제곱한다.

8. void hash\_apply(struct hash\*, hash\_action\_func\*)

parameter: 작업하고자 하는 hash의 주소값과 수행하고자 하는 작업 함수를 인자로 넘긴다.

return: 없음

function: 주어진 hash의 모든 element에 대하여 hash\_action\_func에 주어진 함수를 수행한다. 본 프로젝트에서는 앞서 설계한 square 함수와 triple 함수를 수행하였다.

9. struct hash\_elem\* hash\_delete(struct hash\*, struct hash\_elem\*)

parameter: 작업하고자 하는 hash의 주소값과 삭제하고자 하는 element를 인자로 넘긴다.

return: 삭제된 element를 반환한다.

Function: 주어진 사용자가 hash table과 삭제할 값을 입력하면 주어진 hash table 중에 해당 값을 찾아 그 내용을 모두 삭제한다.

10. struct hash\_elem\* hash\_find(struct hash\*, struct hash\_elem\*)

Parameter: 작업하고자 하는 hash의 주소값과 찾고자 하는 element의 주소값을 인자로 넘긴다.

return: 주어진 element를 hash table에서 찾으면 해당 element를 반환하고, 찾지 못하면 NULL 포인터를 반환한다.

Function: 사용자로부터 hash table의 이름과 찾고자 하는 hash 값을 입력받아 해당 hash에서 해당 데이터를 찾는다. 데이터를 찾으면 해당 데이터를 화면에 출력한다.

11. bool hash\_empty(struct hash\*)

parameter: 비어있는 지 확인하고자 하는 hash 의 주소값을 인자로 넘긴다.

return: 주어진 hash가 비어있으면 true를, 비어있지 않으면 false를 반환한다.

Function: 주어진 hash가 비어있는지 확인한다.

12. size\_t hash\_size(struct hash\*)

parameter: 크기를 알고자 하는 hash의 주소값을 인자로 넘긴다.

return: 주어진 hash에 저장된 element 수를 반환한다.

Function: 주어진 hash의 element 수를 구한다. 결과를 화면에 출력할 때에는 %zu 를 사용한다.

13. void clear(struct hash\_elem\*, void\*)

parameter: 메모리 해제를 원하는 element의 주소값과 NULL 포인터를 인자로 넘긴다.

return: 없음

function: 주어진 element의 메모리를 free 키워드를 통해 해제한다.

14. void hash\_clear(struct hash\*, hash\_action\_func\*)

parameter: 메모리를 초기화하고자 하는 hash의 주소값과 모든 element에 대해 수행할 action 함수를 인자로 받는다.

return: 없음

function: 주어진 hash에 대하여 hash 내부에 선언된 element 메모리를 모두 초기화한다. 이를 위해 hash\_action 함수로 사용된 clear 함수를 미리 정의하였다. 단, hash 내부의 element 중 값을 가지고 있거나 다른 작업에 사용되고 있는 element가 있다면 hash)clear는 작동하지 않는다.

15. unsigned hash\_int\_2(int)

parameter: hash 크기를 구하기 위한 정수형 데이터를 인자로 넘긴다.

return: 주어진 정수형 데이터에 대한 hash 값을 반환한다.

Function: 주어진 정수에 대하여 hash 크기를 구하는 함수이다. 기존의 함수와 구조적으로 크게 달라진 것은 없지만, 기존의 hash\_int가 hash\_bytes 함수를 추가로 호출하는 것에 비해 hash\_int\_2는 추가 함수 호출 없이 곧바로 작업을 처리한다는 점에서 stack activation의 개수를 줄여 메모리 절약과 더불어 프로그램 속도 개선을 조금이나마 기대해 볼 수 있다.

**III. Bitmap**

0. struct bitmap\_with\_name

기존의 bitmap 구조체와 개별 비트맵을 구분할 수 있는 이름을 포함하는 새로운 구조체를 정의하였다. 기존의 list와 hash에서 사용했던 방법처럼 기존의 bitmap 구조체에 이름 멤버를 추가로 할당하려 하였으나 이유를 알 수 없는 에러가 발생해 전체 bitmap 구조체를 감싸는 새로운 구조체를 생성하였다. 10개의 bitmap을 개별적으로 관리하기 위하여 bitmap\_with\_name 이라는 배열을 크기 10으로 설정하였다.

1. Struct bitmap\* bitmap\_create(size\_t)

Parameter: 새로 생성하고자 하는 비트 개수를 인자로 넘긴다.

return: 비트 생성이 정상적으로 완료되면 새로 생성된 비트의 주소값을 반환하고, 메모리 부족 등의 이유로 비트 생성이 이루어지지 않으면 NULL 포인터를 반환한다.

Function: 주어진 개수 만큼의 비트를 생성하고, 초기 값으로 모든 비트에 0을 할당한다.

2. delete

별도의 함수 정의 없이 사용자로부터 delete <bitmap\_name> 을 입력받으면 check\_bitmap을 통해 유효한 비트맵 자료형을 찾고, bitmap\_destroy 함수를 이용해 메모리를 초기화한 후, ds\_bitmap 배열에서 해당 자료형을 삭제한다. 이때, delete 이후에 오는 자료형 이름이 list인지, hash 인지, bitmap인지 구분할 수 없으므로, check\_list와 check\_hash를 먼저 수행한 후 check\_bitmap을 사용하여 확인한다.

3. dumpdata

별도의 함수 정의 없이 메인 함수에서 사용자가 dumpdata <bitmap\_name>을 입력했을 때 check\_bitmap을 이용해 유효한 비트맵 자료형을 찾는다. 이후 bitmap\_size 함수를 이용해 해당 자료형의 비트 크기를 구해 bitmap\_test 함수로 각 비트의 값을 화면에 출력한다. Delete와 마찬가지로 dumpdata 뒤에 오는 자료형 이름이 list인지 hash인지 bitmap인지 알 수 없기 때문에 check\_list, check\_hash의 단계를 거쳐 check+bitmap 함수를 수행한다.

4. void bitmap\_mark(struct bitmap\*, size\_t)

parameter: 해당 작업을 수행할 비트맵 자료형의 주소값과 참으로 표시할 비트 인덱스를 인자로 넘긴다.

return: 없음

function: 주어진 비트맵 자료형에서 주어진 인덱스의 비트를 참(true)로 설정한다.

5. void bitmap\_reset(struct bitmap\*, size\_t)

parameter: 해당 작업을 수행할 비트맵 자료형의 주소값과 거짓으로 표시할 비트 인덱스를 인자로 넘긴다.

return: 없음

function: 주어진 비트맵 자료형에서 주어진 인덱스의 비트를 거짓(false)로 설정한다.

6. void bitmap\_set(struct bitmap\*, size\_t, bool)

parameter: 작업하고자 하는 비트맵 주소값과 값을 변경하고자 하는 비트의 인덱스, 그리고 변경값을 인자로 넘긴다.

return: 없음

function: 주어진 비트맵 자료형에서 주어진 비트의 값을 주어진 이진 값으로 변경한다.

7. void bitmap\_set\_all(struct bitmap\* bool)

Parameter: 작업하고자 하는 비트맵 자료형의 주소값과 셋팅하고자 하는 이진 값을 인자로 넘긴다.

return: 없음

function: 주어진 비트맵 자료형의 모든 비트 값을 주어진 이진 값으로 변경한다.

8. bool bitmap\_test(struct bitmap\*, size\_t)

parameter: 값을 알고자 하는 비트맵 자료형과 해당 비트의 인덱스를 인자로 넘긴다.

return: 주어진 비트맵의 비트의 값을 반환한다.

Function: 사용자가 입력한 비트맵과 비트 수의 값을 화면에 출력한다.

9. size\_t bitmap\_size(struct bitmap\*)

parameter: 비트 수를 알아보고 싶은 비트맵 자료형의 주소값을 인자로 넘긴다.

return: 주어진 비트맵의 비트 수를 반환한다.

Function: 주어진 비트맵에 할당된 비트 수를 알아본다.

10. void bitmap\_set\_multiple(struct bitmap\*, size\_t, size\_t, bool)

parameter: 작업을 수행할 비트맵 자료형의 주소값과 비트의 값을 변경하고자 하는 범위의 첫번째 비트, 그리고 변경하고자 하는 비트의 개수, 마지막으로 초기화값을 인자로 넘긴다.

return: 없음

function: 주어진 비트맵에서 두 번째 인자로 주어진 인덱스부터 세 번째 인자의 개수만큼 네 번째 비트로 값을 바꾼다.

11. bool bitmap\_all(struct bitmap\*, size\_t, size\_t)

parameter: 알아보고자 하는 비트맵 자료형의 주소값과 체크하고자 하는 비트의 범위를 인자로 넘긴다.

return: 주어진 비트 범위의 값이 모두 true 나 false로 같으면 true를 반환하고, 하나라도 다른 비트가 있으면 false를 반환한다.

Function: 주어진 비트맵에서 주어진 범위 안의 비트 값이 모두 같은지 체크하는 함수이다.

12. bool bitmap\_contains(struct bitmap\*, size\_t, size\_t, bool)

parameter: 확인할 비트맵 자료형의 주소값과 비트 값을 체크할 범위, 그리고 해당 범위 내에 특정 값이 있는지 확인하고자 하는 값을 인자로 넘긴다.

return: 주어진 범위 내에 주어진 값이 있므녀 true를, 없으면 false를 반환한다.

function: 주어진 비트맵의 주어진 범위 내에 주어진 값이 있는지 확인한다.

13. bool bitmap\_none(struct bitmap\*, size\_t, size\_t, bool)

parameter: 값을 확인하고자 하는 비트맵 자료형의 주소값과 값을 확인하고자 하는 시작 비트, 끝 비트, 그리고 그 범위 내에 있는지 없는지 확인하고자 하는 값을 인자로 넘긴다.

return: 주어진 범위 내에 주어진 값이 없으면 true를 반환하고, 해당 값이 있으면 false를 반환한다.

Function: 주어진 범위 내에 주어진 값이 있는지 없는지 체크하는 함수이다. Bitmap\_contains와 반대 기능을 수행한다.

14. bool bitmap\_any(struct bitmap\*, size\_t, size\_t)

parameter: 확인하고자 하는 비트맵 자료형의 주소값과 확인하고자 하는 비트의 범위를 넘긴다.

return: 주어진 범위 내에 true 값이 있으면 true를 반환하고, true가 없으면 false를 반환한다.

Function: 주어진 범위 내에 참 값이 있는지 없는지를 확인한다.

15. void bitmap\_dumpdata(struct bitmap\*)

parameter: 확인하고자 하는 비트맵 자료형의 주소값을 인자로 넘긴다.

return: 없음

function: 주어진 비트맵의 비트 값을 16진수로 표시한다. 메인 함수에서 그 값을 화면에 출력할 때에는 unsigned int 자료형을 사용한다.

16. void bitmap\_flip(struct bitmap\*, size\_t)

parameter: 작업을 수행할 비트맵 자료형 주소값과 값을 변경할 비트의 인덱스 값을 인자로 넘긴다.

return: 없음

Function: 주어진 비트맵에서 주어진 인덱스의 비트 값을 바꾼다. 0이었으면 1로, 1이었으면 0으로.

17. size\_t bitmap\_scan(struct bitmap\*, size\_t, size\_t, bool)

parameter: 조사하고자 하는 비트맵 자료형의 주소값과 범위 지정을 위한 두 개의 인덱스, 그리고 알아보고자 하는 값을 인자로 넘긴다.

return: 주어진 범위 내에서 주어진 값이 시작하는 인덱스를 반환한다.

Function: 주어진 비트맵에서 주어진 범위 안의 비트 중 주어진 값을 가진 비트의 위치를 알려주는 함수이다.

18. size\_t bitmap\_scan\_and\_flip(struct bitmap\*, size\_t, size\_t, bool)

parameter: 작업하고자 하는 비트맵 자료형 주소값과 작업을 수행할 비트 범위 지정을 위한 인덱스 두 개, 그리고 값을 인자로 넘긴다.

return: 주어진 범위 내에서 주어진 값이 처음으로 등장하는 위치를 반환한다. 주어진 범위 내에 주어진 값의 비트가 없으면 에러를 반환한다.

Function: 주어진 비트맵에서 주어진 범위 내에 주어진 값의 첫 번째 인덱스를 반환하고, 주어진 범위 내에 주어진 값을 발견하면 모두 반대 값으로 변환한다.

19. size\_t bitmap\_contains(struct bitmap\*, size\_t, size\_t, bool)

parameter: 알아보고자 하는 비트맵 자료형 주소값과 해당 비트맵 내에서 범위를 설정하기 위한 두 인덱스, 그리고 알아보고자 하는 값을 인자로 넘긴다.

return: 주어진 범위 내에 주어진 값이 몇 개 있는지를 반환한다.

Function: 주어진 비트맵에서 주어진 범위 내에 주어진 값이 몇 개 있는지 계산하는 함수이다.

20. struct bitmap\* bitmap\_expand(struct bitmap\*, int)

parameter: 작업하고자 하는 비트맵 자료형 주소값과 해당 비트맵에서 확장하고자 하는 비트의 개수를 인자로 넘긴다.

return: 기존의 비트맵을 성공적으로 확장했다면 확장된 비트맵을 반환하고, 메모리 부족 등의 이유로 확장에 실패했다면 NULL 포인터를 반환한다.

Function: 주어진 비트맵을 주어진 비트만큼 확장한다. Realloc를 이용해 파라미터로 전달받은 비트에 추가로 메모리를 할당한 후, 새로 할당된 비트 수만큼 bitmap\_set 작업을 수행하여 초기 값을 0(false)로 설정한다. Bitmap\_set\_multiple을 사용하면 더욱 쉽게 구현할 수 있을 것으로 보인다.