**Pintos Project 0-2: Pintos Data Structure**

|  |  |
| --- | --- |
| 담당 교수 : | 박성용 교수님 |
| 학번 : | 20141518 |
| 이름 : | 김태헌 |
|  |  |

**반드시 아래의 양식과 순서를 따라서 작성하기 바랍니다.**

1. **Additional Implementation**

|  |  |
| --- | --- |
| **Prototype** | enum operation getOperation(char \*line);  command line의 첫 번째 token을 읽어 operation의 종류 enum 값을 리턴하는 함수입니다. |
| **Parameter** | char \*line  line : command line의 첫 번째 token |
| **Return** | enum operation  operation : operation의 종류 |
| **Function** | enum operation getOperation(char \*line) {  if (strncmp(line, "create", 5) == 0) {  return *CREATE*;  } else if (strncmp(line, "delete", 6) == 0) {  return *DELETE*;  } else if (strncmp(line, "dumpdata", 9) == 0) {  return *DUMP\_DATA*;  } else if (strncmp(line, "list\_", 5) == 0) {  return *LIST\_OPERATION*;  } else if (strncmp(line, "hash\_", 5) == 0) {  return *HASH\_OPERATION*;  } else if (strncmp(line, "bitmap\_", 7) == 0) {  return *BITMAP\_OPERATION*;  } else if (strcmp(line, "quit") == 0) {  return *QUIT*;  } else {  return *DEFAULT*;  } }  입력 operation에 따라 분기문 처리하여 enum타입을 리턴하도록 처리했습니다. |

|  |  |
| --- | --- |
| **Prototype** | struct list \*getListByName(const struct data\_name\_map \*data\_map, const char \*data\_name)  data\_name\_map에 있는 데이터들 중 이름을 기준으로 List를 가져오는 메서드입니다. |
| **Parameter** | const struct data\_name\_map \*data\_map, const char \*data\_name  data\_map : 생성된 자료구조를 담은 테이블, data\_name : 사용자가 입력한 자료 이름 |
| **Return** | struct list \*  list \* : 이름이 일치하는 리스트 포인터 |
| **Function** | struct list \*getListByName(const struct data\_name\_map \*data\_map, const char \*data\_name) {  struct list \*picked\_list = NULL;  for (int i = 0; i < 10; i++) {  if (strcmp(data\_map[i].name, data\_name) == 0) {  picked\_list = data\_map[i].list;  break;  }  }  if (picked\_list == NULL) {  fprintf(stderr, "There is no list name : %s\n", data\_name);  return NULL;  }  return picked\_list; } |

|  |  |
| --- | --- |
| **Prototype** | struct bitmap \*getBitmapByName(const struct data\_name\_map \*name\_table, const char \*data\_name);  data\_name\_map에 있는 데이터들 중 이름을 기준으로 Bitmap를 가져오는 메서드입니다. |
| **Parameter** | const struct data\_name\_map \*data\_map, const char \*data\_name  data\_map : 생성된 자료구조를 담은 테이블, data\_name : 사용자가 입력한 자료 이름 |
| **Return** | struct bitmap \*  bitmap\* : 이름이 일치하는 비트맵포인터 |
| **Function** | struct bitmap \*getBitmapByName(const struct data\_name\_map \*name\_table, const char \*data\_name) {  for (int i = 0; i < 10; i++) {  if (strcmp(name\_table[i].name, data\_name) == 0) {  return name\_table[i].bitmap;  }  }  return NULL; } |

|  |  |
| --- | --- |
| **Prototype** | struct hash \*getHashByName(const struct data\_name\_map \*name\_table, const char \*data\_name);  data\_name\_map에 있는 데이터들 중 이름을 기준으로 Hash를 가져오는 메서드입니다. |
| **Parameter** | const struct data\_name\_map \*data\_map, const char \*data\_name  data\_map : 생성된 자료구조를 담은 테이블, data\_name : 사용자가 입력한 자료 이름 |
| **Return** | struct hash \*  hash\* : 이름이 일치하는 해시 테이블 포인터 |
| **Function** | struct hash \*getHashByName(const struct data\_name\_map \*name\_table, const char \*data\_name) {  for (int i = 0; i < 10; i++) {  if (strcmp(name\_table[i].name, data\_name) == 0) {  return name\_table[i].hash;  }  }  return NULL; } |

|  |  |
| --- | --- |
| **Prototype** | enum data\_structure getDataStructure(const char \*line);  사용자가 create나 delete 시 입력하는 자료구조 이름을 enum으로 변환해주는 메서드입니다. |
| **Parameter** | const char \*line  char \*line : 사용자가 입력한 자료구조 이름 |
| **Return** | enum data\_structure  data\_structure : 자료구조 enum 타입 |
| **Function** | enum data\_structure getDataStructure(const char \*line) {  if (strncmp(line, "list", 4) == 0) {  return *LIST*;  } else if (strncmp(line, "hash", 4) == 0) {  return *HASH*;  } else if (strncmp(line, "bitmap", 6) == 0) {  return *BITMAP*;  } else {  return *NOTHING*;  } } |

|  |  |
| --- | --- |
| **Prototype** | void printConsole(char \*commandLine);  사용자 입력 마다 새로 입력을 초기화 하고, 개행 문자를 제거해주는 메서드 입니다. |
| **Parameter** | char \*commandLine  commandLine : 명령행 사용자 입력 |
| **Return** | void |
| **Function** | void printConsole(char \*commandLine) {  commandLine[0] = '\0';  scanf("%[^\n]", commandLine);  getchar(); } |

|  |  |
| --- | --- |
| **Prototype** | enum list\_operation getListOperation(const char \*line);  사용자가 입력한 list\_method를 enum타입으로 변경해주는 메서드 입니다. |
| **Parameter** | const char \*line  line : 사용자 입력 method token |
| **Return** | enum list\_operation  list\_operation : list 메서드 enum 타입 |
| **Function** | enum list\_operation getListOperation(const char \*line) {  if (strcmp(line, "list\_push\_front") == 0) {  return *LIST\_PUSH\_FRONT*;  } else if (strcmp(line, "list\_push\_back") == 0) {  return *LIST\_PUSH\_BACK*;  } else if (strcmp(line, "list\_pop\_front") == 0) {  return *LIST\_POP\_FRONT*;  } else if (strcmp(line, "list\_pop\_back") == 0) {  return *LIST\_POP\_BACK*;  } else if (strcmp(line, "list\_front") == 0) {  return *LIST\_FRONT*;  } else if (strcmp(line, "list\_back") == 0) {  return *LIST\_BACK*;  } else if (strcmp(line, "list\_insert") == 0) {  return *LIST\_INSERT*;  } else if (strcmp(line, "list\_insert\_ordered") == 0) {  return *LIST\_INSERT\_ORDERED*;  } else if (strcmp(line, "list\_empty") == 0) {  return *LIST\_EMPTY*;  } else if (strcmp(line, "list\_size") == 0) {  return *LIST\_SIZE*;  } else if (strcmp(line, "list\_max") == 0) {  return *LIST\_MAX*;  } else if (strcmp(line, "list\_min") == 0) {  return *LIST\_MIN*;  } else if (strcmp(line, "list\_remove") == 0) {  return *LIST\_REMOVE*;  } else if (strcmp(line, "list\_reverse") == 0) {  return *LIST\_REVERSE*;  } else if (strcmp(line, "list\_shuffle") == 0) {  return *LIST\_SHUFFLE*;  } else if (strcmp(line, "list\_swap") == 0) {  return *LIST\_SWAP*;  } else if (strcmp(line, "list\_sort") == 0) {  return *LIST\_SORT*;  } else if (strcmp(line, "list\_unique") == 0) {  return *LIST\_UNIQUE*;  } else if (strcmp(line, "list\_splice") == 0) {  return *LIST\_SPLICE*;  } else {  return *LIST\_NOTHING*;  } } |

|  |  |
| --- | --- |
| **Prototype** | enum bitmap\_operation getBitmapOperation(const char \*line);  사용자가 입력한 bitmap\_method를 enum타입으로 변경해주는 메서드 입니다. |
| **Parameter** | const char \*line  line: 사용자가 입력한 bitmap\_method 이름 |
| **Return** | enum bitmap\_operation  bitmap\_operation : 비트맵 메서드 enum 타입 |
| **Function** | enum bitmap\_operation getBitmapOperation(const char \*line) {  if (strcmp(line, "bitmap\_mark") == 0) {  return *BITMAP\_MARK*;  } else if (strcmp(line, "bitmap\_all") == 0) {  return *BITMAP\_ALL*;  } else if (strcmp(line, "bitmap\_any") == 0) {  return *BITMAP\_ANY*;  } else if (strcmp(line, "bitmap\_contains") == 0) {  return *BITMAP\_CONTAINS*;  } else if (strcmp(line, "bitmap\_count") == 0) {  return *BITMAP\_COUNT*;  } else if (strcmp(line, "bitmap\_dump") == 0) {  return *BITMAP\_DUMP*;  } else if (strcmp(line, "bitmap\_expand") == 0) {  return *BITMAP\_EXPAND*;  } else if (strcmp(line, "bitmap\_flip") == 0) {  return *BITMAP\_FLIP*;  } else if (strcmp(line, "bitmap\_none") == 0) {  return *BITMAP\_NONE*;  } else if (strcmp(line, "bitmap\_reset") == 0) {  return *BITMAP\_RESET*;  } else if (strcmp(line, "bitmap\_scan") == 0) {  return *BITMAP\_SCAN*;  } else if (strcmp(line, "bitmap\_scan\_and\_flip") == 0) {  return *BITMAP\_SCAN\_AND\_FLIP*;  } else if (strcmp(line, "bitmap\_set") == 0) {  return *BITMAP\_SET*;  } else if (strcmp(line, "bitmap\_set\_all") == 0) {  return *BITMAP\_SET\_ALL*;  } else if (strcmp(line, "bitmap\_set\_multiple") == 0) {  return *BITMAP\_SET\_MULTIPLE*;  } else if (strcmp(line, "bitmap\_size") == 0) {  return *BITMAP\_SIZE*;  } else if (strcmp(line, "bitmap\_test") == 0) {  return *BITMAP\_TEST*;  } else {  return *BITMAP\_NOTHING*;  } } |

|  |  |
| --- | --- |
| **Prototype** | enum hash\_operation getHashOperation(const char \*line)  사용자가 입력한 hash\_method를 enum타입으로 변경해주는 메서드 입니다. |
| **Parameter** | const char \*line  line: 사용자가 입력한 hash\_method 이름 |
| **Return** | enum hash\_operation  hash\_operation : 해시 메서드 enum 타입 |
| **Function** | enum hash\_operation getHashOperation(const char \*line) {  if (strcmp(line, "hash\_apply") == 0) {  return *HASH\_APPLY*;  } else if (strcmp(line, "hash\_clear") == 0) {  return *HASH\_CLEAR*;  } else if (strcmp(line, "hash\_delete") == 0) {  return *HASH\_DELETE*;  } else if (strcmp(line, "hash\_empty") == 0) {  return *HASH\_EMPTY*;  } else if (strcmp(line, "hash\_find") == 0) {  return *HASH\_FIND*;  } else if (strcmp(line, "hash\_insert") == 0) {  return *HASH\_INSERT*;  } else if (strcmp(line, "hash\_replace") == 0) {  return *HASH\_REPLACE*;  } else if (strcmp(line, "hash\_size") == 0) {  return *HASH\_SIZE*;  } else {  return *HASH\_NOTHING*;  } } |

|  |  |
| --- | --- |
| **Prototype** | enum data\_structure getDataTypeByName(const struct data\_name\_map \*name\_table, const char \*data\_name);  data\_name\_map에서 이름을 검색해 데이터 타입을 가져오는 메서드 입니다. |
| **Parameter** | const struct data\_name\_map \*name\_table, const char \*data\_name  name\_table : 생성된 자료구조를 담은 테이블, data\_name : 사용자가 입력한 자료 이름 |
| **Return** | enum data\_structure  data\_structure : 데이터 자료형 |
| **Function** | enum data\_structure getDataTypeByName(const struct data\_name\_map \*name\_table, const char \*data\_name) {  for (int i = 0; i < 10; i++) {  if (strcmp(name\_table[i].name, data\_name) == 0) {  if (name\_table[i].list != NULL) {  return *LIST*;  } else if (name\_table[i].hash != NULL) {  return *HASH*;  } else if (name\_table[i].bitmap != NULL) {  return *BITMAP*;  } else {  continue;  }  }  }  return *NOTHING*; } |

|  |  |
| --- | --- |
| **Prototype** | void createList(struct data\_name\_map name\_table[10], const char \*data\_name, int \*data\_cnt);  list 를 생성하여 name\_table에 저장하는 메서드 입니다. |
| **Parameter** | struct data\_name\_map name\_table[10], const char \*data\_name, int \*data\_cnt  name\_table : 생성된 데이터 풀 , data\_name : 생성할 데이터 이름, data\_cnt : 총 생성된 데이터수 |
| **Return** | void |
| **Function** | void createList(struct data\_name\_map name\_table[10], const char \*data\_name, int \*data\_cnt) {  struct list \*new\_list = (struct list \*) malloc(sizeof(struct list));  list\_init(new\_list);  for (int i = 0; i < 10; i++) {  if (name\_table[i].name[0] == '\0') {  name\_table[i].list = new\_list;  strcpy(name\_table[i].name, data\_name);  break;  }  } } |

|  |  |
| --- | --- |
| **Prototype** | enum data\_structure getDataTypeByName(const struct data\_name\_map \*name\_table, const char \*data\_name);  data\_name\_map에서 이름을 검색해 데이터 타입을 가져오는 메서드 입니다. |
| **Parameter** | const struct data\_name\_map \*name\_table, const char \*data\_name  name\_table : 생성된 자료구조를 담은 테이블, data\_name : 사용자가 입력한 자료 이름 |
| **Return** | enum data\_structure  data\_structure : 데이터 자료형 |
| **Function** | enum data\_structure getDataTypeByName(const struct data\_name\_map \*name\_table, const char \*data\_name) {  for (int i = 0; i < 10; i++) {  if (strcmp(name\_table[i].name, data\_name) == 0) {  if (name\_table[i].list != NULL) {  return *LIST*;  } else if (name\_table[i].hash != NULL) {  return *HASH*;  } else if (name\_table[i].bitmap != NULL) {  return *BITMAP*;  } else {  continue;  }  }  }  return *NOTHING*; } |

|  |  |
| --- | --- |
| **Prototype** | void createList(struct data\_name\_map name\_table[10], const char \*data\_name, int \*data\_cnt);  list 를 생성하여 name\_table에 저장하는 메서드 입니다. |
| **Parameter** | struct data\_name\_map name\_table[10], const char \*data\_name, int \*data\_cnt  name\_table : 생성된 데이터 풀 , data\_name : 생성할 데이터 이름, data\_cnt : 총 생성된 데이터수 |
| **Return** | void |
| **Function** | void createList(struct data\_name\_map name\_table[10], const char \*data\_name, int \*data\_cnt) {  struct list \*new\_list = (struct list \*) malloc(sizeof(struct list));  list\_init(new\_list);  for (int i = 0; i < 10; i++) {  if (name\_table[i].name[0] == '\0') {  name\_table[i].list = new\_list;  strcpy(name\_table[i].name, data\_name);  break;  }  } } |

|  |  |
| --- | --- |
| **Prototype** | void createHash(struct data\_name\_map \*name\_table, const char \*data\_name, int \*data\_cnt);  hash 를 생성하여 name\_table에 저장하는 메서드 입니다. |
| **Parameter** | struct data\_name\_map name\_table[10], const char \*data\_name, int \*data\_cnt  name\_table : 생성된 데이터 풀 , data\_name : 생성할 데이터 이름, data\_cnt : 총 생성된 데이터수 |
| **Return** | void |
| **Function** | void createHash(struct data\_name\_map \*name\_table, const char \*data\_name, int \*data\_cnt) {  struct hash \*new\_hash = (struct hash \*) malloc(sizeof(struct hash));  hash\_hash\_func \*func1 = &hash\_func;  hash\_less\_func \*func2 = &less\_func;  void \*aux = malloc(sizeof(long));  hash\_init(new\_hash, func1, func2, aux);  for (int i = 0; i < 10; i++) {  if (name\_table[i].name[0] == '\0') {  name\_table[i].hash = new\_hash;  strcpy(name\_table[i].name, data\_name);  break;  }  } } |

|  |  |
| --- | --- |
| **Prototype** | void createBitmap(struct data\_name\_map \*name\_table, const char \*data\_name, int bitmap\_size);  bitmap 을 생성하여 name\_table에 저장하는 메서드 입니다. |
| **Parameter** | struct data\_name\_map name\_table[10], const char \*data\_name, int \*data\_cnt  name\_table : 생성된 데이터 풀 , data\_name : 생성할 데이터 이름, data\_cnt : 총 생성된 데이터수 |
| **Return** | void |
| **Function** | void createBitmap(struct data\_name\_map \*name\_table, const char \*data\_name, int bitmap\_size) {  struct bitmap \*new\_bitmap = bitmap\_create(bitmap\_size);  for (int i = 0; i < 10; i++) {  if (name\_table[i].name[0] == '\0') {  name\_table[i].bitmap = new\_bitmap;  strcpy(name\_table[i].name, data\_name);  break;  }  } } |

|  |  |
| --- | --- |
| **Prototype** | void dumpData(const struct data\_name\_map \*data\_name\_table, const char \*data\_name);  데이터 풀에서 name으로 검색해 데이터를 출력하는 메서드 입니다. |
| **Parameter** | const struct data\_name\_map \*data\_name\_table, const char \*data\_name  data\_map : 생성된 자료구조를 담은 테이블, data\_name : 사용자가 입력한 자료 이름 |
| **Return** | void |
| **Function** | void dumpData(const struct data\_name\_map \*data\_name\_table, const char \*data\_name) {  enum data\_structure data\_type = getDataTypeByName(data\_name\_table, data\_name);  struct list \*picked\_list;  struct bitmap \*picked\_bitmap;  struct hash \*picked\_hash;  switch (data\_type) {  case *LIST*:  picked\_list = getListByName(data\_name\_table, data\_name);   if (picked\_list == NULL) {  fprintf(stderr, "There is no list name : %s\n", data\_name);  break;  }  printList(picked\_list);  break;  case *BITMAP*:  picked\_bitmap = getBitmapByName(data\_name\_table, data\_name);   if (picked\_bitmap == NULL) {  fprintf(stderr, "There is no bitmap name : %s\n", data\_name);  break;  }  printBitmap(picked\_bitmap);  break;  case *HASH*:  picked\_hash = getHashByName(data\_name\_table, data\_name);   if (picked\_hash == NULL) {  fprintf(stderr, "There is no hash name : %s\n", data\_name);  break;  }  printHash(picked\_hash);  break;  case *NOTHING*:  fprintf(stderr, "There is no type name : %s\n", data\_name);  break;  default:  fprintf(stderr, "Unexpected Error : %s\n", data\_name);  break;  } } |

|  |  |
| --- | --- |
| **Prototype** | void printList(struct list \*picked\_list);  list를 출력하는 메서드 입니다. |
| **Parameter** | struct list \*picked\_list  picked\_list : 사용자가 선택한 리스트 |
| **Return** | void |
| **Function** | void printList(struct list \*picked\_list) {  if (list\_empty(picked\_list)) {  return;  }  struct list\_elem \*elem = list\_begin(picked\_list);  while (elem != NULL) {  if (elem->next == NULL) {  break;  }  struct list\_item \*pItem = list\_entry(elem, struct list\_item, elem);  printf("%d ", pItem->data);  elem = elem->next;  }  printf("\n"); } |

|  |  |
| --- | --- |
| **Prototype** | void printBitmap(struct bitmap \*pBitmap);  Bitmap을 출력해주는 메서드 입니다. |
| **Parameter** | struct bitmap \*pBitmap  pBitmap : 사용자가 선택한 비트맵 포인터. |
| **Return** | void |
| **Function** | void printBitmap(struct bitmap \*pBitmap) {  int bit\_cnt = (int) pBitmap->bit\_cnt;   for (int j = 0; j <= bit\_cnt / ELEM\_BITS; j++) {  unsigned long value = (unsigned long) pBitmap->bits[j];  for (int i = 0; j \* 64 + i < bit\_cnt && i < 64; i++) {  printf("%lu", value % 2);  value /= 2;  }  }   printf("\n"); } |

|  |  |
| --- | --- |
| **Prototype** | void printHash(struct hash \*pHash);  Hash를 출력해주는 메서드 입니다. |
| **Parameter** | struct hash \*pHash  pHash : 사용자가 선택한 hash 포인터 |
| **Return** | void |
| **Function** | void printHash(struct hash \*pHash) {  if (hash\_empty(pHash)) {  return;  }  for (int i = 0; i < pHash->bucket\_cnt; i++) {  struct list \*list = &pHash->buckets[i];  if (list\_empty(list)) {  continue;  }  struct list\_elem \*elem = list\_begin(list);  while (elem != NULL) {  if (elem->next == NULL) {  break;  }  struct list\_item \*pItem = list\_entry(elem, struct list\_item, elem);  printf("%d ", pItem->data);  elem = elem->next;  }  }  printf("\n"); } |

|  |  |
| --- | --- |
| **Prototype** | void listPushFront(struct list \*picked\_list, const char \*data);  list\_push\_front를 수행하는 메서드입니다. |
| **Parameter** | struct list \*picked\_list, const char \*data  picked\_list: 사용자가 선택한 리스트, data : push 할 값 |
| **Return** | void |
| **Function** | void listPushFront(struct list \*picked\_list, const char \*data) {  struct list\_elem \*new\_elem = (struct list\_elem \*) malloc(sizeof(struct list\_item));  struct list\_item \*pItem = list\_entry(new\_elem, struct list\_item, elem);  pItem->data = (int) strtol(data, NULL, 10);  list\_push\_front(picked\_list, new\_elem); } |

|  |  |
| --- | --- |
| **Prototype** | void listPushBack(struct list \*picked\_list, const char \*data);  list\_push\_back을 수행해주는 메서드입니다. |
| **Parameter** | struct list \*picked\_list, const char \*data  picked\_list : 사용자가 선택한 리스트, data : push할 데이터 |
| **Return** | void |
| **Function** | void listPushBack(struct list \*picked\_list, const char \*args2) {  struct list\_elem \*new\_elem = (struct list\_elem \*) malloc(sizeof(struct list\_item));  struct list\_item \*pItem = list\_entry(new\_elem, struct list\_item, elem);  pItem->data = (int) strtol(args2, NULL, 10);  list\_push\_back(picked\_list, new\_elem); } |

|  |  |
| --- | --- |
| **Prototype** | void processListOperation(const struct data\_name\_map \*data\_name\_table, const char \*operation, char \*args1, char \*args2,  char \*args3);  list 관련 operation을 수행해주는 메서드입니다. |
| **Parameter** | const struct data\_name\_map \*data\_name\_table, const char \*operation, char \*args1, char \*args2,  char \*args3  data\_name\_table : 데이터 풀, operation : 명령어 args1 ~ args3 : 사용자 명령 argument |
| **Return** | void |
| **Function** | void processListOperation(const struct data\_name\_map \*data\_name\_table, const char \*operation, char \*args1, char \*args2,  char \*args3) {  enum data\_structure dataType = getDataTypeByName(data\_name\_table, args1);  if (dataType != *LIST*) {  fprintf(stderr, "%s is not a List. Please try again.\n", args1);  return;  }  struct list \*pickedList = getListByName(data\_name\_table, args1);   switch (getListOperation(operation)) {  case *LIST\_PUSH\_FRONT*:  listPushFront(pickedList, args2);  break;  case *LIST\_PUSH\_BACK*:  listPushBack(pickedList, args2);  break;  case *LIST\_POP\_FRONT*:  list\_pop\_front(pickedList);  break;  case *LIST\_POP\_BACK*:  list\_pop\_back(pickedList);  break;  case *LIST\_FRONT*: {  struct list\_elem \*pElem = list\_front(pickedList);  struct list\_item \*pItem = list\_entry(pElem, struct list\_item, elem);  printf("%d\n", pItem->data);  break;  }  case *LIST\_BACK*: {  struct list\_elem \*pElem = list\_back(pickedList);  struct list\_item \*pItem = list\_entry(pElem, struct list\_item, elem);  printf("%d\n", pItem->data);  break;  }  case *LIST\_INSERT*: {  struct list\_elem \*before = list\_begin(pickedList);  struct list\_elem \*pElem = (struct list\_elem \*) malloc(sizeof(struct list\_item));  struct list\_item \*pItem = list\_entry(pElem, struct list\_item, elem);  pItem->data = (int) strtol(args3, NULL, 10);   int index = (int) strtol(args2, NULL, 10);  if (index == 0) {  list\_push\_front(pickedList, pElem);  } else {  for (int i = 0; i < index; i++) {  before = list\_next(before);  }  list\_insert(before, pElem);  }  break;  }  case *LIST\_INSERT\_ORDERED*: {  struct list\_elem \*pElem = (struct list\_elem \*) malloc(sizeof(struct list\_item));  struct list\_item \*pItem = list\_entry(pElem, struct list\_item, elem);  pItem->data = (int) strtol(args2, NULL, 10);  bool (\*less\_func)() = list\_less;  list\_insert\_ordered(pickedList, pElem, less\_func, NULL);  break;  }  case *LIST\_EMPTY*: {  printf(list\_empty(pickedList) ? "true\n" : "false\n");  break;  }  case *LIST\_SIZE*: {  printf("%zu\n", list\_size(pickedList));  break;  }  case *LIST\_MAX*: {  bool (\*less\_func)() = list\_less;  struct list\_elem \*pElem = list\_max(pickedList, less\_func, NULL);  struct list\_item \*pItem = list\_entry(pElem, struct list\_item, elem);   printf("%d\n", pItem->data);  break;  }  case *LIST\_MIN*: {  bool (\*less\_func)() = list\_less;  struct list\_elem \*pElem = list\_min(pickedList, less\_func, NULL);  struct list\_item \*pItem = list\_entry(pElem, struct list\_item, elem);   printf("%d\n", pItem->data);  break;  }  case *LIST\_REMOVE*: {  int remove\_index = (int) strtol(args2, NULL, 10);  if (remove\_index >= list\_size(pickedList)) {  fprintf(stderr, "Remove index must smaller than last index.\n");  break;  }  struct list\_elem \*iter = list\_begin(pickedList);   if (remove\_index == 0) {  list\_remove(iter);  } else {  for (int i = 0; i < remove\_index; i++) {  iter = list\_next(iter);  }  list\_remove(iter);  }  break;  }  case *LIST\_REVERSE*:  list\_reverse(pickedList);  break;  case *LIST\_SHUFFLE*:  list\_shuffle(pickedList);  break;  case *LIST\_SWAP*: {  int a = (int) strtol(args2, NULL, 10);  int b = (int) strtol(args3, NULL, 10);  struct list\_elem \*pA = list\_get(pickedList, a);  struct list\_elem \*pB = list\_get(pickedList, b);  list\_swap(pA, pB);  break;  }  case *LIST\_SORT*: {  bool (\*less\_func)() = list\_less;  list\_sort(pickedList, less\_func, NULL);  break;  }  case *LIST\_UNIQUE*: {  bool (\*less\_func)() = list\_less;  if (args2 == NULL) {  list\_unique(pickedList, NULL, less\_func, NULL);  } else {  struct list \*second\_list = getListByName(data\_name\_table, args2);  list\_unique(pickedList, second\_list, less\_func, NULL);  }  break;  }  case *LIST\_SPLICE*: {  char \*args4 = strtok(NULL, " ");  char \*args5 = strtok(NULL, " ");  int before\_idx = (int) strtol(args2, NULL, 10);  int start\_add\_idx = (int) strtol(args4, NULL, 10);  int end\_add\_idx = (int) strtol(args5, NULL, 10);  struct list \*second\_list = getListByName(data\_name\_table, args3);   struct list\_elem \*pBefore = list\_get(pickedList, before\_idx);  struct list\_elem \*pStart = list\_get(second\_list, start\_add\_idx);  struct list\_elem \*pEnd = list\_get(second\_list, end\_add\_idx);  list\_splice(pBefore, pStart, pEnd);  break;  }  case *LIST\_NOTHING*: {  fprintf(stderr, "Unsupported List operation. pleas try again\n");  break;  }   default:  fprintf(stderr, "Unsupported list operation : %s\n", operation);  break;  } } |

|  |  |
| --- | --- |
| **Prototype** | bool list\_less(struct list\_elem \*elem1, struct list\_elem \*elem2, void \*aux);  list 대소 비교 메서드 입니다. |
| **Parameter** | struct list\_elem \*elem1, struct list\_elem \*elem2, void \*aux  elem1,2 : 리스트 엘리먼트, aux : 보조 포인터 |
| **Return** | bool  elem1<elem2 결과 값 |
| **Function** | bool list\_less(struct list\_elem \*elem1, struct list\_elem \*elem2, void \*aux) {  struct list\_item \*pItem1 = list\_entry(elem1, struct list\_item, elem);  struct list\_item \*pItem2 = list\_entry(elem2, struct list\_item, elem);   return pItem1->data < pItem2->data; } |

|  |  |
| --- | --- |
| **Prototype** | void init\_table(struct data\_name\_map \*data\_name\_table);  데이터 풀을 초기화 해주는 메서드입니다. |
| **Parameter** | struct data\_name\_map \*data\_name\_table  data\_name\_table : 데이터 풀 |
| **Return** | void |
| **Function** | void init\_table(struct data\_name\_map \*data\_name\_table) {  for (int i = 0; i < 10; i++) {  data\_name\_table[i].name[0] = '\0';  data\_name\_table[i].list = NULL;  data\_name\_table[i].bitmap = NULL;  data\_name\_table[i].hash = NULL;  } } |

|  |  |
| --- | --- |
| **Prototype** | void create\_data(int \*data\_cnt, struct data\_name\_map \*data\_name\_table, const char \*type\_name, const char \*data\_name,  const char \*args3)  data 타입에 맞는 데이터를 생성해주는 메서드입니다. |
| **Parameter** | int \*data\_cnt, struct data\_name\_map \*data\_name\_table, const char \*type\_name, const char \*data\_name,  const char \*args3  data\_cnt : 총 데이터 생성 수 , data\_name\_table: 데이터 풀, type\_name : 데이터 타입 이름, data\_name: 데이터 변수 이름, args3 : 추가 argument |
| **Return** | void |
| **Function** | void create\_data(int \*data\_cnt, struct data\_name\_map \*data\_name\_table, const char \*type\_name, const char \*data\_name,  const char \*args3) {  switch (getDataStructure(type\_name)) {  case *LIST*:  createList(data\_name\_table, data\_name, data\_cnt);  break;  case *HASH*:  createHash(data\_name\_table, data\_name, data\_cnt);  break;  case *BITMAP*: {  int bitmap\_size = (int) strtol(args3, NULL, 10);   createBitmap(data\_name\_table, data\_name, bitmap\_size);  break;  }  case *NOTHING*:  fprintf(stderr, "There is no type : %s\n", type\_name);  break;  }  (\*data\_cnt)++; } |

|  |  |
| --- | --- |
| **Prototype** | void delete\_data(int \*data\_cnt, struct data\_name\_map \*data\_name\_table, const char \*args1);  변수명으로 데이터를 지우는 메서드 입니다. |
| **Parameter** | int \*data\_cnt, struct data\_name\_map \*data\_name\_table, const char \*args1  data\_cnt : 데이터 총 개수 , data\_name\_table : 데이터 풀, args1 : 데이터 변수 명 |
| **Return** | void |
| **Function** | void delete\_data(int \*data\_cnt, struct data\_name\_map \*data\_name\_table, const char \*args1) {  for (int i = 0; i < 10; i++) {  if (strcmp(data\_name\_table[i].name, args1) == 0) {  if (data\_name\_table[i].list != NULL) {  free(data\_name\_table[i].list);  } else if (data\_name\_table[i].bitmap != NULL) {  free(data\_name\_table[i].bitmap);  } else if (data\_name\_table[i].hash != NULL) {  free(data\_name\_table[i].hash);  }  data\_name\_table[i].name[0] = '\0';  break;  }  }  (\*data\_cnt)--; } |

|  |  |
| --- | --- |
| **Prototype** | void processBitmapOperation(struct data\_name\_map data\_name\_table[10], char \*operation, char \*args1, char \*args2,  char \*args3);  Bitmap 메서드를 수행하는 메서드입니다. |
| **Parameter** | struct data\_name\_map data\_name\_table[10], char \*operation, char \*args1, char \*args2,  char \*args3  data\_name\_table: 데이터 풀, operation : 메서드 이름, args1~3 : 추가 명령 argument |
| **Return** | void |
| **Function** | void processBitmapOperation(struct data\_name\_map data\_name\_table[10], char \*operation, char \*args1, char \*args2,  char \*args3) {  enum data\_structure dataType = getDataTypeByName(data\_name\_table, args1);  if (dataType != *BITMAP*) {  fprintf(stderr, "%s is not a Bitmap. Please try again.\n", args1);  return;  }  struct bitmap \*pickedBitmap = getBitmapByName(data\_name\_table, args1);   switch (getBitmapOperation(operation)) {  case *BITMAP\_MARK*: {  size\_t idx = (size\_t) strtol(args2, NULL, 10);  bitmap\_mark(pickedBitmap, idx);  break;  }  case *BITMAP\_ALL*: {  size\_t start = (size\_t) strtol(args2, NULL, 10);  size\_t end = (size\_t) strtol(args3, NULL, 10);  printBool(bitmap\_all(pickedBitmap, start, end));  break;  }  case *BITMAP\_ANY*: {  size\_t start = (size\_t) strtol(args2, NULL, 10);  size\_t end = (size\_t) strtol(args3, NULL, 10);  printBool(bitmap\_any(pickedBitmap, start, end));  break;  }  case *BITMAP\_CONTAINS*: {  size\_t start = (size\_t) strtol(args2, NULL, 10);  size\_t cnt = (size\_t) strtol(args3, NULL, 10);  bool tf = getBoolean(strtok(NULL, " "));  printBool(bitmap\_contains(pickedBitmap, start, cnt, tf));  break;  }  case *BITMAP\_COUNT*: {  size\_t start = (size\_t) strtol(args2, NULL, 10);  size\_t cnt = (size\_t) strtol(args3, NULL, 10);  bool tf = getBoolean(strtok(NULL, " "));  printf("%zu\n", bitmap\_count(pickedBitmap, start, cnt, tf));  break;  }  case *BITMAP\_DUMP*: {  bitmap\_dump(pickedBitmap);  break;  }  case *BITMAP\_EXPAND*: {  size\_t size = (size\_t) strtol(args2, NULL, 10);  struct bitmap \*pBitmap = bitmap\_expand(pickedBitmap, size);  for (int i = 0; i < 10; i++) {  if (strcmp(data\_name\_table[i].name, args1) == 0) {  data\_name\_table->bitmap = pBitmap;  break;  }  }  break;  }  case *BITMAP\_FLIP*: {  size\_t idx = (size\_t) strtol(args2, NULL, 10);  bitmap\_flip(pickedBitmap, idx);  break;  }  case *BITMAP\_NONE*: {  size\_t start = (size\_t) strtol(args2, NULL, 10);  size\_t cnt = (size\_t) strtol(args3, NULL, 10);  printf(bitmap\_none(pickedBitmap, start, cnt) ? "true\n" : "false\n");  }  case *BITMAP\_RESET*: {  size\_t idx = (size\_t) strtol(args2, NULL, 10);  bitmap\_reset(pickedBitmap, idx);  break;  }  case *BITMAP\_SCAN*: {  size\_t start = (size\_t) strtol(args2, NULL, 10);  size\_t cnt = (size\_t) strtol(args3, NULL, 10);  bool tf = getBoolean(strtok(NULL, " "));  printf("%zu\n", bitmap\_scan(pickedBitmap, start, cnt, tf));  break;  }  case *BITMAP\_SCAN\_AND\_FLIP*: {  size\_t start = (size\_t) strtol(args2, NULL, 10);  size\_t cnt = (size\_t) strtol(args3, NULL, 10);  bool tf = getBoolean(strtok(NULL, " "));  printf("%zu\n", bitmap\_scan\_and\_flip(pickedBitmap, start, cnt, tf));  break;  }  case *BITMAP\_SET*: {  size\_t idx = (size\_t) strtol(args2, NULL, 10);  bool tf = getBoolean(args3);  bitmap\_set(pickedBitmap, idx, tf);  break;  }  case *BITMAP\_SET\_ALL*: {  bool tf = getBoolean(args2);  bitmap\_set\_all(pickedBitmap, tf);  break;  }  case *BITMAP\_SET\_MULTIPLE*: {  size\_t start = (size\_t) strtol(args2, NULL, 10);  size\_t cnt = (size\_t) strtol(args3, NULL, 10);  bool tf = getBoolean(strtok(NULL, " "));  bitmap\_set\_multiple(pickedBitmap, start, cnt, tf);  break;  }  case *BITMAP\_SIZE*: {  printf("%zu\n", bitmap\_size(pickedBitmap));  break;  }  case *BITMAP\_TEST*: {  size\_t idx = (size\_t) strtol(args2, NULL, 10);  printBool(bitmap\_test(pickedBitmap, idx));  break;  }  case *BITMAP\_NOTHING*: {  fprintf(stderr, "Unsupported Bitmap operation. pleas try again\n");  break;  }  default:  fprintf(stderr, "Unexpected Error. please try again\n");  break;  } } |

|  |  |
| --- | --- |
| **Prototype** | void processHashOperation(struct data\_name\_map data\_name\_table[10], char \*operation, char \*args1, char \*args2, char \*args3)  hash 메서드를 수행해주는 메서드입니다. |
| **Parameter** | struct data\_name\_map data\_name\_table[10], char \*operation, char \*args1, char \*args2, char \*args3  data\_name\_table : 데이터 풀 , operation: 메서드 이름, args1~3 : 추가 명령 argument |
| **Return** | void |
| **Function** | void processHashOperation(struct data\_name\_map data\_name\_table[10], char \*operation, char \*args1, char \*args2, char \*args3) {  enum data\_structure dataType = getDataTypeByName(data\_name\_table, args1);  if (dataType != *HASH*) {  fprintf(stderr, "%s is not a HashTable. Please try again.\n", args1);  return;  }  struct hash \*pickedHash = getHashByName(data\_name\_table, args1);   switch (getHashOperation(operation)) {  case *HASH\_INSERT*: {  int data = (int) strtol(args2, NULL, 10);  struct hash\_elem \*pElem = (struct hash\_elem \*) malloc(sizeof(struct hash\_item));  struct hash\_item \*pItem = hash\_entry(pElem, struct hash\_item, elem);  pItem->data = data;  hash\_insert(pickedHash, pElem);  break;  }  case *HASH\_SIZE*: {  printf("%zu\n", hash\_size(pickedHash));  break;  }  case *HASH\_EMPTY*: {  printBool(hash\_empty(pickedHash));  break;  }  case *HASH\_CLEAR*: {  hash\_clear(pickedHash, &delete\_elem\_func);  break;  }  case *HASH\_FIND*: {  int data = (int) strtol(args2, NULL, 10);  size\_t size = pickedHash->bucket\_cnt;  bool flag = false;  for (int i = 0; i < size; i++) {  struct list\_elem \*elem = list\_begin(&pickedHash->buckets[i]);  while (elem != NULL) {  if (elem->next == NULL) {  break;  }  struct list\_item \*pItem = list\_entry(elem, struct list\_item, elem);  if (pItem->data == data) {  flag = true;  break;  }  elem = elem->next;  }  if (flag) {  break;  }  }  if (flag) {  printf("%d\n", data);  }  break;  }  case *HASH\_REPLACE*: {  int data = (int) strtol(args2, NULL, 10);  struct hash\_elem \*pElem = (struct hash\_elem \*) malloc(sizeof(struct hash\_item));  struct hash\_item \*pItem = hash\_entry(pElem, struct hash\_item, elem);  pItem->data = data;  hash\_replace(pickedHash, pElem);  break;  }  case *HASH\_DELETE*: {  int data = (int) strtol(args2, NULL, 10);  size\_t size = pickedHash->bucket\_cnt;  for (int i = 0; i < size; i++) {  bool flag = false;  struct list\_elem \*elem = list\_begin(&pickedHash->buckets[i]);  while (elem != NULL) {  struct list\_item \*pItem = list\_entry(elem, struct list\_item, elem);  if (pItem->data == data) {  hash\_delete(pickedHash, list\_elem\_to\_hash\_elem(elem));  flag = true;  break;  }  elem = elem->next;  }  if (flag) {  break;  }  }  break;  }  case *HASH\_APPLY*: {  if (strcmp(args2, "square") == 0) {  hash\_apply(pickedHash, &square);  } else if (strcmp(args2, "triple") == 0) {  hash\_apply(pickedHash, &triple);  } else if (strcmp(args2, "-10") == 0) {  hash\_apply(pickedHash, &minus10);  } else {  fprintf(stderr, "Unknown function. Please try again");  break;  }  break;  }  case *HASH\_NOTHING*: {  fprintf(stderr, "Unsupported Hash Table operation. pleas try again\n");  break;  }  default:  fprintf(stderr, "Unexpected Error. please try again\n");  break;  } } |

|  |  |
| --- | --- |
| **Prototype** | bool getBoolean(char \*args);  String을 bool 값으로 변경해주는 메서드입니다. |
| **Parameter** | char \*args  args : string bool 값 |
| **Return** | bool  변환된 bool 값 |
| **Function** | bool getBoolean(char \*args) {  bool tf;  if (strcmp(args, "true") == 0) {  tf = true;  } else {  tf = false;  }  return tf; } |

|  |  |
| --- | --- |
| **Prototype** | void printBool(bool result);  bool을 출력해주는 메서드 입니다. |
| **Parameter** | bool result  result : 출력할 bool 값 |
| **Return** | void |
| **Function** | void printBool(bool result) { printf(result ? "true\n" : "false\n"); } |

|  |  |
| --- | --- |
| **Prototype** | void square(struct hash\_elem \*elem, void \*aux);  hash element에 square 해주는 메서드입니다. |
| **Parameter** | struct hash\_elem \*elem, void \*aux  elem: apply 대상, aux: 보조 포인터 |
| **Return** | void |
| **Function** | void square(struct hash\_elem \*elem, void \*aux) {  struct hash\_item \*pItem = hash\_entry(elem, struct hash\_item, elem);  int data = pItem->data;  pItem->data = data \* data; } |

|  |  |
| --- | --- |
| **Prototype** | void triple(struct hash\_elem \*elem, void \*aux)  hash element에 triple 해주는 메서드입니다. |
| **Parameter** | struct hash\_elem \*elem, void \*aux  elem: apply 대상, aux: 보조 포인터 |
| **Return** | void |
| **Function** | void triple(struct hash\_elem \*elem, void \*aux) {  struct hash\_item \*pItem = hash\_entry(elem, struct hash\_item, elem);  int data = pItem->data;  pItem->data = data \* data \* data; } |

|  |  |
| --- | --- |
| **Prototype** | void minus10(struct hash\_elem \*elem, void \*aux)  hash element에 -10 해주는 메서드입니다. |
| **Parameter** | struct hash\_elem \*elem, void \*aux  elem: apply 대상, aux: 보조 포인터 |
| **Return** | void |
| **Function** | void minus10(struct hash\_elem \*elem, void \*aux) {  struct hash\_item \*pItem = hash\_entry(elem, struct hash\_item, elem);  int data = pItem->data;  pItem->data = data - 10; } |

|  |  |
| --- | --- |
| **Prototype** | unsigned hash\_func(const struct hash\_elem \*e, void \*aux);  hash 값을 구해주는 메서드입니다. |
| **Parameter** | const struct hash\_elem \*e, void \*aux  e : 해쉬 엘리먼트, aux: 보조 포인터 |
| **Return** | unsigned |
| **Function** | unsigned hash\_func(const struct hash\_elem \*e, void \*aux) {  struct hash\_item \*pItem = hash\_entry(e, struct hash\_item, elem);  return hash\_int(pItem->data); } |

|  |  |
| --- | --- |
| **Prototype** | bool less\_func(const struct hash\_elem \*a, const struct hash\_elem \*b, void \*aux);  hash less function. hash element 비교 메서드입니다. |
| **Parameter** | const struct hash\_elem \*a, const struct hash\_elem \*b, void \*aux  a, b: 비교 대상 해쉬 엘리먼트, aux : 보조 포인터 |
| **Return** | bool  변환된 bool 값 |
| **Function** | bool less\_func(const struct hash\_elem \*a, const struct hash\_elem \*b, void \*aux) {  struct hash\_item \*pItemA = hash\_entry(a, struct hash\_item, elem);  struct hash\_item \*pItemB = hash\_entry(b, struct hash\_item, elem);   return pItemA->data < pItemB->data; } |

|  |  |
| --- | --- |
| **Prototype** | void delete\_elem\_func(struct hash\_elem \*e, void \*aux);  hash 엘리먼트 삭제 메서드입니다. |
| **Parameter** | struct hash\_elem \*e, void \*aux  e : 삭제할 엘리먼트, aux : 보조 포인터 |
| **Return** | void |
| **Function** | void delete\_elem\_func(struct hash\_elem \*e, void \*aux) {  struct hash\_item \*pItem = hash\_entry(e, struct hash\_item, elem);  free(pItem); } |

1. **List**

|  |  |
| --- | --- |
| **Prototype** | void list\_shuffle(struct list \*list)  리스트를 무작위로 섞는 메서드입니다. |
| **Parameter** | struct list \*list  섞을 대상이 되는 리스트 포인터 |
| **Return** | void |
| **Function** | /\* Shuffles the order of LIST. \*/ void list\_shuffle(struct list \*list) {  if (!list\_empty(list)) {  srand((unsigned int) time(NULL));  unsigned int count = (unsigned) list\_size(list);  for (unsigned int i = 0; i < count; i++) {  int random\_num1 = rand() % count;  int random\_num2 = rand() % count;  struct list\_elem \*pA = list\_get(list, random\_num1);  struct list\_elem \*pB = list\_get(list, random\_num2);  list\_swap(pA, pB);  }  } }  리스트 원소 개수만큼 랜덤한 두 개의 엘리먼트를 집어서 서로 스왑하는 방식으로 섞습니다. 랜덤 값은 time을 seed로 하는 rand함수를 통해 구합니다. |

|  |  |
| --- | --- |
| **Prototype** | struct list\_elem \* list\_get(struct list \*list, int index)  index에 위치한 list element를 구하는 메서드입니다. 매 번 구현하기 번거로워서 구현했습니다. |
| **Parameter** | struct list \*list, int index  list : element를 찾을 대상 리스트, index : element index |
| **Return** | struct list\_elem \*  주어진 idx에 해당하는 element포인터. 없는 경우 null |
| **Function** | struct list\_elem \* list\_get(struct list \*list, int index) {  if (list\_size(list) <= index) {  fprintf(stderr, "Index must be smaller than size\n");  return NULL;  }  if (index == 0) {  return list\_begin(list);  } else {  struct list\_elem \*pElem = list\_begin(list);  for (int i = 0; i < index; i++) {  pElem = list\_next(pElem);  }  return pElem;  } }  list의 사이즈보다 입력한 idx 값이 클 경우 에러와 NULL을 반환합니다. index가 0인 경우 begin 포인터를 반환하고 나머지의 경우 travel을 통해 탐색하여 가져옵니다. |

|  |  |
| --- | --- |
| **Prototype** | void list\_swap(struct list\_elem \*a, struct list\_elem \*b)  리스트 상의 두 element의 순서를 바꾸는 메서드입니다. |
| **Parameter** | struct list\_elem \*a, struct list\_elem \*b  a,b : 순서를 바꿀 대상 element |
| **Return** | void |
| **Function** | void list\_swap(struct list\_elem \*a, struct list\_elem \*b) {  struct list\_item \*pItemA = list\_entry(a, struct list\_item, elem);  struct list\_item \*pItemB = list\_entry(b, struct list\_item, elem);  int tmp = pItemA->data;  pItemA->data = pItemB->data;  pItemB->data = tmp; }  두 포인터의 값을 바꾸는 식으로 swap을 구현했습니다. 기존엔 포인터 자체의 위치를 바꾸고자 했으나, 값만 바꾸는 게 훨씬 간단하다고 판단되어 수정했습니다. |

1. **Hash Table**

|  |  |
| --- | --- |
| **Prototype** | unsigned hash\_int2 (int i)  int에 대한 hash값을 리턴해주는 메서드입니다. |
| **Parameter** | int i  i : hash값을 계산할 정수 값 |
| **Return** | unsigned  hash 값 |
| **Function** | unsigned hash\_int2 (int i) {  return (unsigned)(i/100); }  예제에서 주어진 것 처럼 주어진 정수값을 100으로 나누는 방식으로 구현했습니다. |

1. **Bitmap**

|  |  |
| --- | --- |
| **Prototype** | struct bitmap \* bitmap\_expand(struct bitmap \*bitmap, int size)  비트맵을 주어진 사이즈만큼 확장하는 메서드입니다. |
| **Parameter** | struct bitmap \*bitmap, int size  bitmap: 확장시킬 비트맵, size: 확장시킬 사이즈 |
| **Return** | struct bitmap \*  확장된 비트맵 |
| **Function** | /\* Expand Bitmap by size \*/ struct bitmap \* bitmap\_expand(struct bitmap \*bitmap, int size) {  ASSERT(bitmap != NULL);    size\_t before\_elem\_cnt = elem\_cnt(bitmap->bit\_cnt);  size\_t after\_elem\_cnt = elem\_cnt(size) + before\_elem\_cnt;   if (size <= 0) {  fprintf(stderr, "Size must be positive integer.\n");  return NULL;  }   elem\_type \*after\_elem = (elem\_type \*) realloc(bitmap->bits, sizeof(elem\_type) \* after\_elem\_cnt);  bitmap->bit\_cnt = size + bitmap->bit\_cnt;  bitmap->bits = after\_elem;  return bitmap; }  확장 될 사이즈의 element count를 계산하여 영억을 reallocation 받습니다. C의 realloc은 기존의 데이터를 보존한다고 하여, 새로 할당받지 않았습니다. bitmap이 NULL인 경우 Assertion Error를 반환하고, size가 0인 경우 NULL을 반환합니다. |