
ASSIGNMENT#1

자료구조 02 분반

20220357

구민정

1. Assignment #1-1: Implementation

1) 헤더파일 선언

A. 동적배열에 관한 헤더파일

```
#ifndef _DYNAMICARRAY_H_
#define __DYNAMICARRAY_H_
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct dynamicArray {
    int* array; // 배열 포인터 선언
    int curr; // 현재 배열의 size(element의 개수)
    int capacity; // 배열의 용량
}dArray;

void initArray(dArray* arr, int initialCap);
bool ArrayIsEmpty(dArray* arr);
bool ArrayisFull(dArray* arr);
void addIndex(dArray* arr, int index, int item);
void deleteIndex(dArray* arr, int index);
void accessIndex(dArray* arr, int index);
void printarr(dArray* arr);
void freeDynamicArray(dArray* arr);

#endif
```

B. 연결리스트에 관한 헤더파일

```
#ifndef _LINKEDLIST_H_
#define __LINKEDLIST_H_
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct listNode* listPointer; // 아직 선언 안된 자료형에 대해 포인터 참조만 가능
typedef struct {
    int data;
    int cnt;
    listPointer start;
    listPointer link;
}listNode; // self referential structure

void initList(listNode* list);
bool ListIsEmpty(listNode* list);
void addNode(listNode* list, int index, int data);
void deleteNode(listNode* list, int index);
void accessHead(listNode* list);
void printNode(listNode* list);
void freeLinkedList(listNode* list);

#endif
```

2) 함수 구현

A. 동적 배열 함수 구현

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
//DynamicArray 구조체 선언
typedef struct dynamicArray {
    int* array; // 배열 포인터 선언
    int curr; // 현재 배열의 size(element의 개수)
    int capacity; // 배열의 용량
}dArray;
//초기화
void initArray(dArray* arr, int initialCap) {
```

```

arr->array = (int*)malloc(initialCap * sizeof(int));
arr->curr = 0;
arr->capacity = initialCap;
}
//동적배열이 비어있는지 판단
bool ArrayisEmpty(dArray* arr) {
    if (arr->curr == 0) return true;
    else return false;
}
//동적배열이 가득 찼는지 판단
bool ArrayisFull(dArray* arr) {
    if (arr->curr != 0 && (arr->curr == arr->capacity)) return true;
    else return false;
}
//index 와 item 을 인자로 받아 동적배열에 추가하는 함수
void addIndex(dArray* arr, int index, int item) {
    //배열이 가득 차 있다면 용량을 2 배로 증가
    if (ArrayisFull(arr)) {
        arr->capacity *= 2;
        arr->array = realloc(arr->array, (arr->capacity) * sizeof(int));
        printf("Array is full. Multiplized capacity : %d\n", arr->capacity);
    }
    //인자로 받은 index 가 유효하지 않은 index 인 경우
    if (index < 0 || index > arr->curr) {
        printf("해당 index 에는 데이터를 추가할 수 없습니다.\n");
        return;
    }
    //index 가 맨앞, 특정 index 인 경우
    if (index < arr->curr) {
        for (int i = arr->curr; i > index; i--) { // 뒤에서부터 이동
            arr->array[i] = arr->array[i - 1];
        }
    }
    //모든 경우에 대해 배열이 정렬되었으므로 item 추가
    arr->array[index] = item;
    (arr->curr)+=1;
}
//index 를 인자로 받아 해당 데이터를 삭제하는 함수
void deleteIndex(dArray* arr, int index) {
    //비어있는 경우
    if (ArrayisEmpty(arr)) {
        printf("array is empty.\n");
        return;
    }
    //유효하지 않은 index 인 경우
    if (index < 0 || index >= arr->curr) {
        printf("해당 index 에는 데이터가 없습니다.\n");
        return;
    }
    //index 의 데이터를 삭제하기 위해 뒤의 데이터들을 당겨옴
    for (int i = index; i < arr->curr - 1; i++) {
        arr->array[i] = arr->array[i + 1];
    }
    (arr->curr)--=1;
    //삭제 후에 배열의 size 를 2 분의 1 로 줄일 수 있다면 배열 size 조정
    if ((arr->curr) < ((arr->capacity) / 2 + 1) && (arr->capacity > 2)) {
        arr->capacity /= 2;
        arr->array = realloc(arr->array, (arr->capacity) * sizeof(int)); // 배열 크기
        printf("Resized capacity : %d\n", arr->capacity);
    }
}
//index 를 인자로 받아 접근하고 데이터를 출력하는 함수
void accessIndex(dArray* arr, int index) {
    if (index >= 0 && index < arr->curr) {
        for (int i = 0; i <= arr->curr; i++) {
            if (i == index) {
                printf("Element in index %d : %d\n", index, arr->array[index]);
            }
        }
        return;
    }
}

```

```

    }
}
//유효하지 않은 index 인 경우
printf("해당 index 에는 데이터가 존재하지 않습니다.\n");
}
//배열을 출력하는 함수
void printarr(dArray* arr) {
    if (ArrayIsEmpty(arr)) {
        printf("array is empty.\n");
        return;
    }
    printf(">>");
    for (int i = 0; i < arr->curr; i++) {
        printf("%d ", arr->array[i]);
    }
    printf("\n");
}
//배열을 free
void freeDynamicArray(dArray* arr) {
    if (arr->array != NULL) { // 배열이 할당되어 있는지 확인
        free(arr->array); // 배열 메모리 해제
        arr->array = NULL; // 배열 포인터를 NULL 로 설정
    }
}
}

```

B. 연결리스트 함수 구현

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
//ListNode 구조체 선언
typedef struct listNode* listPointer; // 아직 선언 안된 자료형에 대해 포인터 참조만 가능
typedef struct {
    int data;
    int cnt;
    listPointer start;
    listPointer link;
}listNode; // self referential structure
//초기화
void initList(listNode* list) {
    list->start = (listNode*)malloc(sizeof(listNode*));
    list->cnt = 0;
}
//list 가 비어있는 경우
bool listIsEmpty(listNode* list) {
    if (list->cnt<=0) return true;
    else return false;
}
//index 와 data 를 인자로 받아와 list 에 추가
void addNode(listNode* list, int index, int data) {
    //list 가 비어있는 경우 새로운 노드 생성
    if (listIsEmpty(list)){
        listNode* newNode = (listNode*)malloc(sizeof(listNode*));
        list->start = newNode;
        newNode->data = data;
        newNode->link = NULL;
        list->cnt++;
    }
    //비어있지는 않지만 유효하지 않은 index 인 경우
    else if (list->cnt + 1 <= index) {
        printf("잘못된 위치입니다.\n");
        return;
    }
    //유효한 index 이지만 첫번째 노드가 아닌 경우
    else {
        listNode* curr = list->start;
        for (int i = 1; i < index; i++) {
            curr = curr->link;
        }
    }
}

```

```

        listNode* addNode = (listNode*)malloc(sizeof(listNode*));
        addNode->data = data;
        addNode->link = curr->link;
        curr->link = addNode;
        list->cnt++;
    }
}
//index 를 인자로 받아 해당 노드 삭제
void deleteNode(listNode* list, int index) {
    listNode* curr = list->start;
    listNode* prev = NULL;
    //유효하지 않은 index 인 경우
    for (int i = 0; i < index; i++) {
        prev = curr;
        curr = curr->link;
        if (curr == NULL) {
            printf("잘못된 인덱스입니다.\n");
            return;
        }
    }
    // 삭제할 노드가 첫 번째 노드인 경우
    if (prev == NULL) {
        list->start = curr->link;
    }
    //중간, 마지막 노드인 경우
    else {
        prev->link = curr->link;
    }
    free(curr);
    list->cnt--;
}
//head 에 접근하는 함수
void accessHead(listNode* list) {
    if (ListIsEmpty(list)) {
        printf("리스트가 비어 있습니다.\n");
        return;
    }
    listNode* first = list->start;
    printf("Head element of the list : %d\n", first->data);
}
//list 를 출력하는 함수
void printNode(listNode* list) {
    if (ListIsEmpty(list)) {
        printf("list 가 비어있습니다.\n");
        return;
    }
    listNode* curr = list->start;
    for (curr = list->start; curr != NULL; curr = curr->link) {
        printf("%d ", curr->data);
    }
    printf("\n");
}
//list free
void freeLinkedList(listNode* list) {
    listNode* curr = list->start;
    while (curr != NULL) {
        listNode* temp = curr;
        curr = curr->link;
        free(temp);
    }
}
}

```

3) Main 작성

```

#include "DynamicArray.h"
#include "LinkedList.h"
int main() {
    //DynamicArray 와 LinkedList 를 테스트
    printf("<DynamicArray Test>\n");
    dArray arr;
}

```

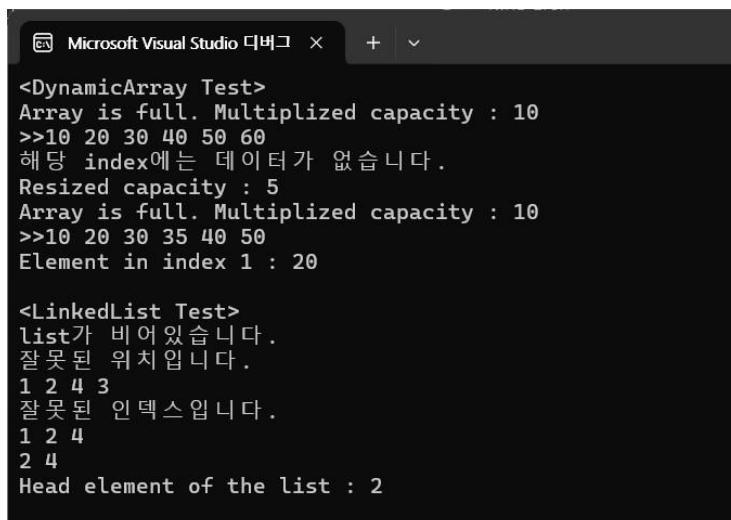
```

initArray(&arr, 5); // 초기 용량이 5 인 동적 배열 생성
// 요소 추가
addIndex(&arr, 0, 10); // 배열의 인덱스 0 에 요소 추가
addIndex(&arr, 1, 20); // 배열의 인덱스 1 에 요소 추가
addIndex(&arr, 2, 30); // 배열의 인덱스 2 에 요소 추가
addIndex(&arr, 3, 40); // 배열의 인덱스 3 에 요소 추가
addIndex(&arr, 4, 50); // 배열의 인덱스 4 에 요소 추가
addIndex(&arr, 5, 60); // 배열의 인덱스 5 에 요소 추가 --> size 조정[10]
// 배열 출력
printarr(&arr); // >> 10 20 30 40 50 60
// 요소 삭제, 추가
deleteIndex(&arr, 6); // 배열의 인덱스 6 에서 요소 삭제 --> 유효하지 않은 index
deleteIndex(&arr, 5); // 배열의 인덱스 5 에서 요소 삭제 --> 10 20 30 40 50 (size 절반으로
재조정[5])
addIndex(&arr, 3, 35); // 배열의 인덱스 2 에 요소 추가 --> size 조정[10]
// 배열 출력
printarr(&arr); // >> 10 20 30 35 40 50
// 배열에서 특정 인덱스의 요소 출력
accessIndex(&arr, 1); // Element in index 1 : 20
// 배열의 메모리 해제
freeDynamicArray(&arr);
printf("\n");
printf("<LinkedList Test>\n");
listNode l;
initList(&l); //list 생성
//빈 list 출력
printNode(&l);
//node 추가
addNode(&l, 0, 1); //리스트의 인덱스 0 에 요소 추가
addNode(&l, 2, 2); //유효하지 않은 인덱스
addNode(&l, 1, 2); //리스트의 인덱스 1 에 요소 추가
addNode(&l, 2, 3); //리스트의 인덱스 2 에 요소 추가
addNode(&l, 2, 4); //리스트의 인덱스 2 에 요소 추가
printNode(&l); //1 2 4 3

//node 삭제
deleteNode(&l, 4); //유효하지 않은 인덱스
deleteNode(&l, 3); //3 삭제
printNode(&l); //1 2 4
deleteNode(&l, 0); //1 삭제
printNode(&l); //2 4
//head 접근
accessHead(&l); //2 출력
//리스트의 메모리 해제
freeLinkedList(&l);
return 0;
}

```

4) Test 결과



```

Microsoft Visual Studio 디버그
<DynamicArray Test>
Array is full. Multiplized capacity : 10
>>10 20 30 40 50 60
해당 index에는 데이터가 없습니다.
Resized capacity : 5
Array is full. Multiplized capacity : 10
>>10 20 30 35 40 50
Element in index 1 : 20

<LinkedList Test>
list가 비어있습니다.
잘못된 위치입니다.
1 2 4 3
잘못된 인덱스입니다.
1 2 4
2 4
Head element of the list : 2

```

2. Assignment #1-2: Time Complexity Analysis

1) 헤더파일 생성

```
#ifndef _TIMECOMPLEXITY_H_
#define _TIMECOMPLEXITY_H_
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>

// 동적 배열에 대한 각 연산의 실행 시간을 측정하는 함수
void measureDynamicArrayTime(dArray* arr, int size);
// 연결 리스트에 대한 각 연산의 실행 시간을 측정하는 함수
void measureLinkedListTime(listNode* list, int size);

#endif
```

2) 시간복잡도 계산 함수 구현

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <windows.h> // Windows 에서 시간 측정을 위해 추가
#include "DynamicArray.h"
#include "LinkedList.h"
#include "TimeComplexity.h"

//size 가 작은 경우 초단위일 때 0 초로 나타나는 경우가 있어 밀리초로 변환 후 출력
// 동적 배열에 대한 각 연산의 실행 시간을 측정하는 함수
void measureDynamicArrayTime(dArray* arr, int size) {
    LARGE_INTEGER frequency, start, end;
    double time_taken;
    QueryPerformanceFrequency(&frequency); // Performance Counter 의 주파수를 가져옴
    // insertion 측정
    QueryPerformanceCounter(&start);
    for (int i = 0; i < size; i++) {
        addIndex(arr, i, i);
    }
    QueryPerformanceCounter(&end);
    time_taken = ((double)(end.QuadPart - start.QuadPart)) / frequency.QuadPart * 1000.0; //
    밀리초로 변환
    printf("Dynamic Array Insertion Time (size %d): %f milliseconds\n", size, time_taken);
    // deletion 측정
    QueryPerformanceCounter(&start);
    for (int i = 0; i < size; i++) {
        deleteIndex(arr, i);
    }
    QueryPerformanceCounter(&end);
    time_taken = ((double)(end.QuadPart - start.QuadPart)) / frequency.QuadPart * 1000.0; //
    밀리초로 변환
    printf("Dynamic Array Deletion Time (size %d): %f milliseconds\n", size, time_taken);
    // access 측정
    for (int i = 0; i < size; i++) {
        addIndex(arr, i, i);
    }
    QueryPerformanceCounter(&start);
    for (int i = 0; i < size; i++) {
        accessIndex(arr, i);
    }
    QueryPerformanceCounter(&end);
    time_taken = ((double)(end.QuadPart - start.QuadPart)) / frequency.QuadPart * 1000.0; //
    밀리초로 변환
    printf("Dynamic Array Access Time (size %d): %f milliseconds\n", size, time_taken);
}

// 연결 리스트에 대한 각 연산의 실행 시간을 측정하는 함수
void measureLinkedListTime(listNode* list, int size) {
    LARGE_INTEGER frequency, start, end;
    double time_taken;
    QueryPerformanceFrequency(&frequency); // Performance Counter 의 주파수를 가져옴
```

```

// insertion 측정
QueryPerformanceCounter(&start);
for (int i = 0; i < size; i++) {
    addNode(list, i, i);
}
QueryPerformanceCounter(&end);
time_taken = ((double)(end.QuadPart - start.QuadPart)) / frequency.QuadPart * 1000.0; //
밀리초로 변환
printf("Linked List Insertion Time (size %d): %f milliseconds\n", size, time_taken);
// deletion 측정
QueryPerformanceCounter(&start);
for (int i = 0; i < size; i++) {
    deleteNode(list, 0);
}
QueryPerformanceCounter(&end);
time_taken = ((double)(end.QuadPart - start.QuadPart)) / frequency.QuadPart * 1000.0; //
밀리초로 변환
printf("Linked List Deletion Time (size %d): %f milliseconds\n", size, time_taken);
// access 측정
for (int i = 0; i < size; i++) {
    addNode(list, i, i);
}
QueryPerformanceCounter(&start);
for (int i = 0; i < size; i++) {
    accessHead(list);
}
QueryPerformanceCounter(&end);
time_taken = ((double)(end.QuadPart - start.QuadPart)) / frequency.QuadPart * 1000.0; //
밀리초로 변환
printf("Linked List Access Time (size %d): %f milliseconds\n", size, time_taken);
}

```

3) 측정 결과

```

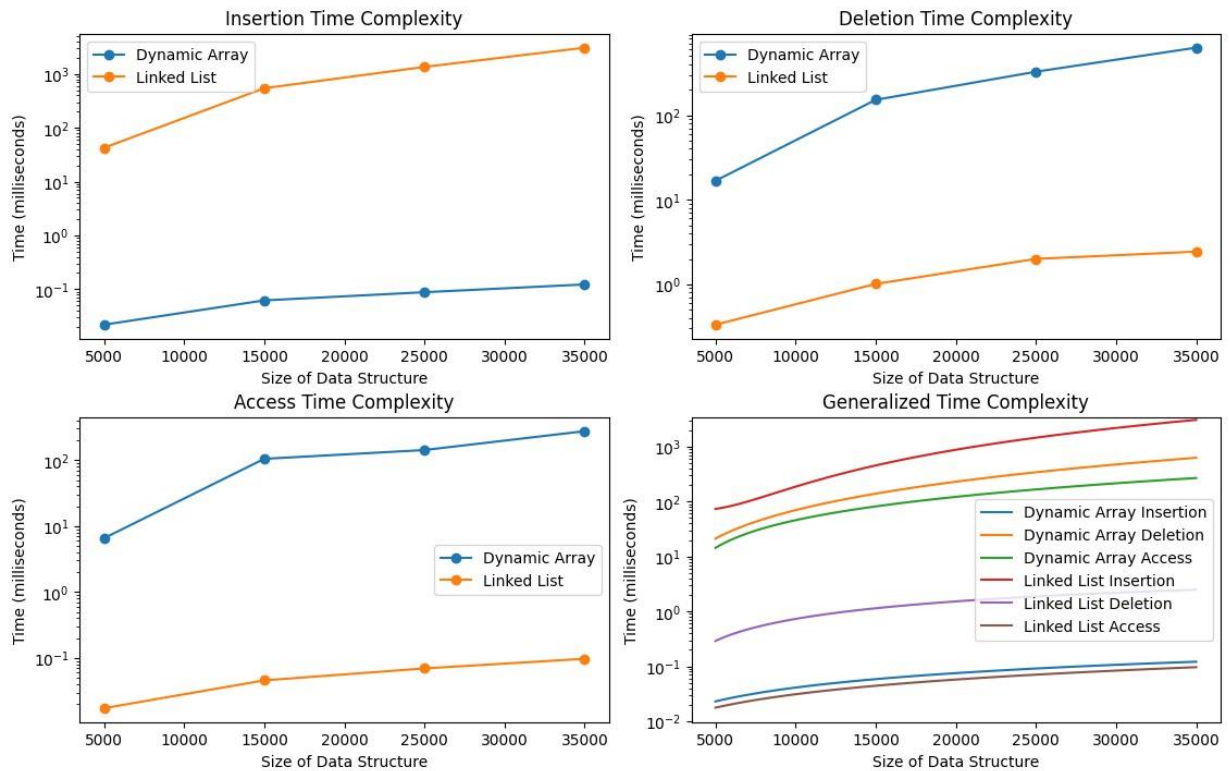
Microsoft Visual Studio 디버그
Measurements for size: 5000
Dynamic Array Insertion Time (size 5000): 0.021900 milliseconds
Dynamic Array Deletion Time (size 5000): 16.773100 milliseconds
Dynamic Array Access Time (size 5000): 6.545800 milliseconds
Linked List Insertion Time (size 5000): 42.541900 milliseconds
Linked List Deletion Time (size 5000): 0.333200 milliseconds
Linked List Access Time (size 5000): 0.017300 milliseconds

Measurements for size: 15000
Dynamic Array Insertion Time (size 15000): 0.062000 milliseconds
Dynamic Array Deletion Time (size 15000): 151.984500 milliseconds
Dynamic Array Access Time (size 15000): 104.695900 milliseconds
Linked List Insertion Time (size 15000): 544.563800 milliseconds
Linked List Deletion Time (size 15000): 1.011900 milliseconds
Linked List Access Time (size 15000): 0.045700 milliseconds

Measurements for size: 25000
Dynamic Array Insertion Time (size 25000): 0.088000 milliseconds
Dynamic Array Deletion Time (size 25000): 326.764400 milliseconds
Dynamic Array Access Time (size 25000): 142.362600 milliseconds
Linked List Insertion Time (size 25000): 1356.039300 milliseconds
Linked List Deletion Time (size 25000): 2.000000 milliseconds
Linked List Access Time (size 25000): 0.069400 milliseconds

Measurements for size: 35000
Dynamic Array Insertion Time (size 35000): 0.122700 milliseconds
Dynamic Array Deletion Time (size 35000): 628.676200 milliseconds
Dynamic Array Access Time (size 35000): 275.097700 milliseconds
Linked List Insertion Time (size 35000): 3090.379800 milliseconds
Linked List Deletion Time (size 35000): 2.436500 milliseconds
Linked List Access Time (size 35000): 0.097100 milliseconds

```

4) 분석

A. 동적 배열

● 삽입/삭제 (Insertion/Deletion)

삽입/삭제(Insertion/Deletion) 시간 복잡도는 $O(n)$ 이다. 특정 인덱스에 데이터를 삽입, 삭제하기 위해 기존의 데이터를 옮기는 작업이 필요하다. 예를 들어, 0번째 인덱스에 데이터를 삽입하기 위해서는 현재 배열의 0번째 인덱스부터 마지막 인덱스까지의 데이터들을 모두 한 인덱스 뒤로 옮겨야 한다. 따라서 최악의 경우 시간 복잡도는 $O(n)$ 이다.

● 접근(Access)

접근(Access)의 시간 복잡도는 $O(1)$ 이다. 따라서 배열의 가장 큰 장점은 인덱스에 곧바로 접근할 수 있다는 것이다.

● 탐색(Search)

탐색(Search)의 시간 복잡도는 $O(n)$ 이다. 0번째 인덱스부터 마지막 인덱스까지 접근하면서 원하는 데이터를 찾는다. 배열 안에 찾는 데이터가 없거나 또는 찾으려는 데이터가 마지막에 있는 경우 n 개의 데이터를 살펴봐야 한다. 따라서 최악의 경우 시간 복잡도는 $O(n)$ 이다.

B. 연결 리스트

● 삽입/삭제 (Insertion/Deletion)

삽입/삭제(Insertion/Deletion) 시간 복잡도는 $O(1)$ 이다. 삽입, 삭제할 노드의 주변 노드들의 Link만 수정하면 되기 때문이다. 따라서 삽입, 삭제가 실행되는 시간은 특정 값에 비례하지 않고 항상 일정하다. 직접 측정된 데이터에서는 삽입, 삭제할 노드를 탐색하는 과정이 필요하기 때문에 최악의 경우 $O(n)$ 의 시간 복잡도가 필요하다. 하지만 연결 리스트의 맨 앞에 삽입하는 경우는 이미 Head를 알고 있기 때문에 $O(1)$ 의 시간 복잡도를 가진다. 또한 Tail을 알고 있다면, 맨 뒤에 삽입하는 경우 $O(1)$ 의 시간 복잡도를 가진다.

- 접근 및 탐색 (Access/Search)

접근 및 탐색 (Access/Search)의 시간 복잡도는 $O(n)$ 이다. 가장 앞 노드부터 다음 노드를 하나씩 보면서 원하는 데이터를 갖는 데이터를 찾는다. 리스트 안에 찾는 데이터가 없거나 또는 찾으려는 데이터가 마지막 노드에 있는 경우 n 개의 노드를 전부 탐색해야 한다. 따라서 최악의 경우 시간 복잡도는 $O(n)$ 이다.

3. Assignment #1-3: Space Complexity Analysis

1) 헤더파일 생성

```
#define _SPACECOMPLEXITY_H_
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// 동적 배열의 공간 복잡도를 계산하는 함수
int calculateDynamicArraySpaceComplexity(int size, int num_elements);
// 연결 리스트의 공간 복잡도를 계산하는 함수
int calculateLinkedListSpaceComplexity(int num_elements);
// 동적 배열의 데이터가 저장된 공간의 메모리만 계산하는 함수
int calculateDynamicArrayDataSpace(int num_elements);
// 연결 리스트의 데이터가 저장된 공간의 메모리만 계산하는 함수
int calculateLinkedListDataSpace(int num_elements);

#endif
```

2) 시간복잡도 계산 함수 구현

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "DynamicArray.h"
#include "LinkedList.h"

// 동적 배열의 공간 복잡도를 계산하는 함수
// 추가 구조적 데이터를 고려
int calculateDynamicArraySpaceComplexity(int size, int num_elements) {
    // 배열의 크기 = 요소의 개수 * sizeof(int)
    // + 배열의 구조적 데이터를 저장하기 위해 sizeof(dArray)
    int array_size = num_elements * sizeof(int) + sizeof(dArray);
    return array_size;
}

// 연결 리스트의 공간 복잡도를 계산하는 함수
// 각 노드마다 데이터와 포인터 저장
int calculateLinkedListSpaceComplexity(int num_elements) {
    // 각 노드의 크기 = 데이터의 크기
    // + 다음 노드를 가리키는 포인터의 크기
    int node_size = sizeof(int) + sizeof(listNode*);
    // 총 연결 리스트의 크기 = 각 노드의 크기 * 요소의 개수
    int linked_list_size = node_size * num_elements;
    return linked_list_size;
}

// 동적 배열의 데이터가 저장된 공간의 메모리만 계산하는 함수
int calculateDynamicArrayDataSpace(int num_elements) {
    // 데이터가 저장된 공간 = 요소의 개수 * 데이터 형식의 크기(int)
    int data_space = num_elements * sizeof(int);
    return data_space;
}

// 연결 리스트의 데이터가 저장된 공간의 메모리만 계산하는 함수
int calculateLinkedListDataSpace(int num_elements) {
    // 데이터가 저장된 공간 = 요소의 개수 * 데이터 형식의 크기(int)
    int data_space = num_elements * sizeof(int);
    return data_space;
}
```

3) 측정 결과

```

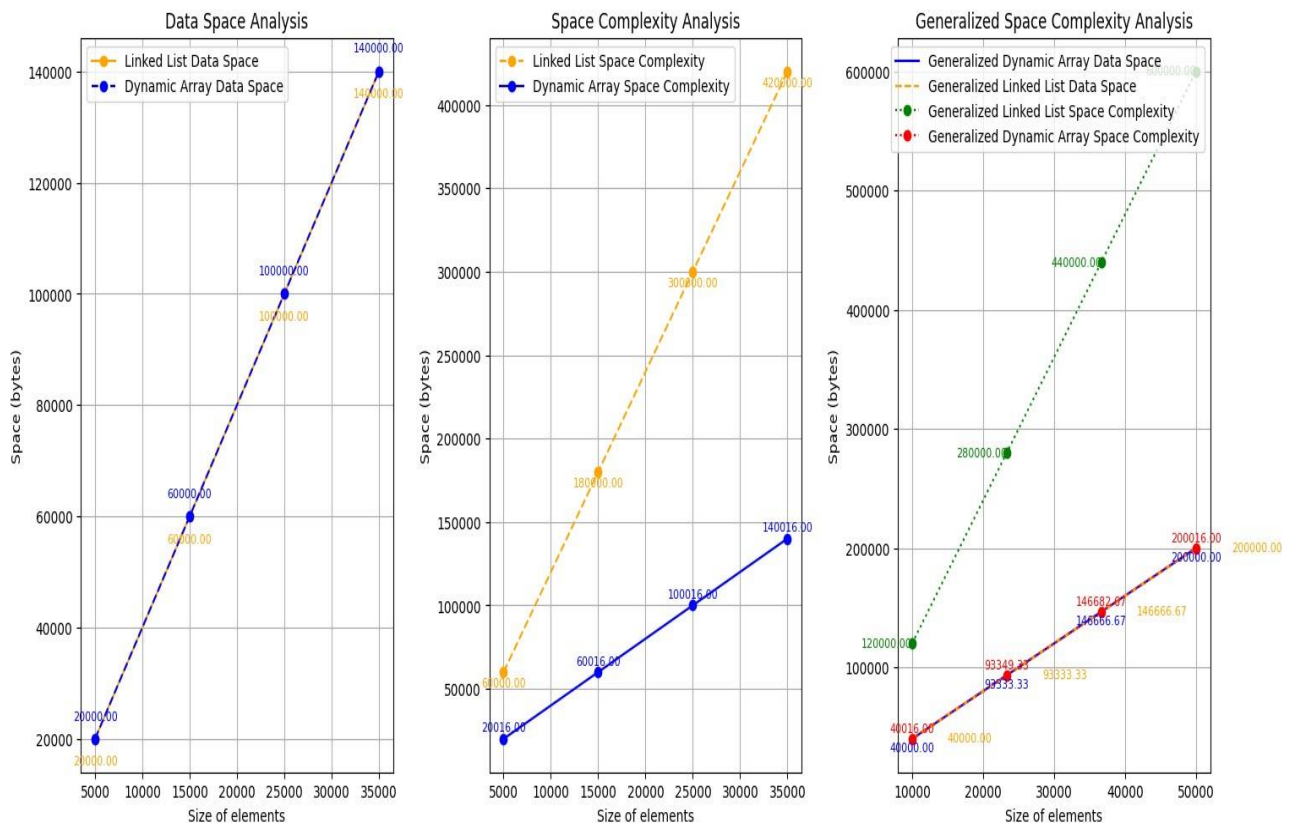
Microsoft Visual Studio 디버그
Dynamic Array Data Space (size 5000, elements 5000): 20000 bytes
Dynamic Array Space Complexity (size 5000, elements 5000): 20016 bytes
Linked List Data Space (size 5000, elements 5000): 20000 bytes
Linked List Space Complexity (size 5000, elements 5000): 60000 bytes

Dynamic Array Data Space (size 15000, elements 15000): 60000 bytes
Dynamic Array Space Complexity (size 15000, elements 15000): 60016 bytes
Linked List Data Space (size 15000, elements 15000): 60000 bytes
Linked List Space Complexity (size 15000, elements 15000): 180000 bytes

Dynamic Array Data Space (size 25000, elements 25000): 100000 bytes
Dynamic Array Space Complexity (size 25000, elements 25000): 100016 bytes
Linked List Data Space (size 25000, elements 25000): 100000 bytes
Linked List Space Complexity (size 25000, elements 25000): 300000 bytes

Dynamic Array Data Space (size 35000, elements 35000): 140000 bytes
Dynamic Array Space Complexity (size 35000, elements 35000): 140016 bytes
Linked List Data Space (size 35000, elements 35000): 140000 bytes
Linked List Space Complexity (size 35000, elements 35000): 420000 bytes

```



4) 분석

- 동적 배열

동적 배열의 공간 복잡도는 데이터 요소의 크기와 배열의 크기에 따라 결정된다. 데이터를 저장하는 배열 자체만을 본다면, 포인터 등 추가적인 메모리 사용이 없으므로 데이터의 개수와 거의 동일한 공간 복잡도를 가진다. 예를 들어 측정된 데이터를 보면, 크기가 5000일 때 동적 배열의 데이터 공간은 20000 bytes이고 공간 복잡도는 20016 bytes이다.

- 연결 리스트

연결 리스트의 공간 복잡도는 포인터의 개수와 데이터 요소의 크기에 따라 결정된다. 각 노드마다 포인터가 필요하므로, 크기가 증가할수록 공간이 선형적으로 증가한다. 측정된 데이터처럼, 크기가 5000일 때 연결 리스트의 데이터 공간은 20000 bytes이고 공간 복잡도는 60000 bytes이다.

4. 결론

동적 배열은 데이터 접근 측면에서 연결 리스트보다 빠른 성능을 보이지만, 크기가 커질수록 데이터의 삽입과 삭제에서는 재할당이 빈번하게 발생해 시간 복잡도가 증가한다. 공간 복잡도 측면에서는 데이터 외 추가적인 메모리는 거의 사용하지 않지만, 경우에 따라 배열의 빈공간이 많아 비효율적일 수 있다. 반면에 연결 리스트는 탐색과 조회 연산에서 동적 배열보다 느린 경향이 있지만, 크기에 무관하게 상수 시간의 삽입 및 삭제가 가능하다. 또한, 메모리 관리 측면에서 동적으로 할당되는 노드들로 이루어져 있어 메모리 낭비가 적다. 따라서, 데이터 접근(탐색, 조회)이 빈번할 경우에는 Dynamic Array를, 데이터 수정(삽입, 삭제)이 빈번할 경우 Linked List를 활용하는 것이 적합하다.

한편, 각 자료 구조는 서로 다른 특징과 장단점을 가지기 때문에 상황에 맞는 자료구조를 선택하는 것이 중요하다. 적절한 자료구조의 선택은 메모리 사용의 효율성을 높이고, 데이터 접근과 처리 속도를 개선한다. 따라서 프로그램에 필요한 자료구조를 적절히 선택하는 것은 데이터 관리를 통한 효율적인 코드 작성과 더 나은 프로그램 설계에 영향을 미친다.