다양한 자료 구조를 이용한 LRU Simulator의 구현

이 문서에서는 LRU Simulator을 구현하는 방법에 대해 다룬다. 다양한 언어와 자료 구조를 통하여 LRU Simulator를 구현하고, 그 실행 결과를 비교하고 있다.

파이썬을 통한 LRU Simulator의 구현

리스트를 통한 구현

CacheSimulator 클래스의 메서드를 알아보자.

```
class CacheSimulator:
    def __init__(self, cache_slots):
        self.cache_slots = cache_slots
        self.cache = []
        self.cache_hit = 0
        self.tot_cnt = 0

    def do_sim(self, page):
        self.tot_cnt += 1
        if page in self.cache:
            self.cache.remove(page)
            self.cache_hit += 1
        elif len(self.cache) >= self.cache_slots:
            self.cache.pop(0)
        self.cache.append(page)
        # Do programming here!

    def print_stats(self):
        print("cache_slot = ", self.cache_slots, "cache_hit = ", self.cache_hit,
"hit ratio = ", self.cache_hit / self.tot_cnt)
```

생성자 `__init__(cache_slots)`

`self.cache_slots`는 캐시 슬롯의 크기를 정의하는 속성이다. 인자로 받은 `cache_slots`을 그 값으로 설정한다.

`self.cache[]`는 캐시 데이터의 집합을 정의하는 속성이다.

`self.cache_hit`은 캐시의 저장 공간에 접근한 횟수를 정의하는 속성이다.

`self.tot_cnt`는 총 저장장치 접근 횟수를 정의하는 속성이다.

시뮬레이션 실행 메서드 `do_sim()`

`page`를 불러오는 상황을 가정하여 cache hit이 되었다면 `self.cache_hit`을 1 증가시키는 메서드이다.

이 메서드의 작동 절차를 알아보자.

1. page를 받아 접근할 데이터를 넘겨받는다.
2. 총 시도 횟수인 self.tot_cnt를 1 증가시킨다.
3. 캐시(self.cache) 안에 page의 값이 있는 경우
    1. self.cache에서 page를 삭제한다.
    2. 이후 캐시 이용 횟수(self.cache_hit)를 1 증가시킨다.
4. 캐시(self.cache) 안에 page의 값이 없고, 캐시(self.cache)의 길이가 캐시 슬롯의 크기가 self.cache_slots보다 큰 경우
    1. self.cache의 마지막 값을 pop한다.
5. self.cache의 끝에 page를 append 한다.

결과 출력 메서드 print_stats()

이 메서드는 캐시 슬롯의 크기(self.cache_slots), 캐시 저장 공간에 접근한 횟수(self.cache_hit), 총 시도 중에서 캐시 저장 공간에 접근한 비율((self.cache_hit)/(self.tot_cnt))을 차례로 출력하는 메서드이다.

출력 결과

```
$ python3 lru_sim_array.py
cache_slot =  100 cache_hit =  14554 hit ratio =  0.14554
cache_slot =  200 cache_hit =  15377 hit ratio =  0.15377
cache_slot =  300 cache_hit =  16001 hit ratio =  0.16001
cache_slot =  400 cache_hit =  16665 hit ratio =  0.16665
cache_slot =  500 cache_hit =  17628 hit ratio =  0.17628
cache_slot =  600 cache_hit =  18796 hit ratio =  0.18796
cache_slot =  700 cache_hit =  20387 hit ratio =  0.20387
cache_slot =  800 cache_hit =  23947 hit ratio =  0.23947
cache_slot =  900 cache_hit =  26340 hit ratio =  0.2634
cache_slot =  1000 cache_hit =  28110 hit ratio =  0.2811
```

cache_slot이 10배 늘어나는 동안 hit_ratio가 2배 늘어났다. 따라서 cache_slot과 hit_ratio는 양의 상관관계를 가진다고 추측할 수 있다.

순환 연결 리스트(Circular Linked List)를 통한 구현

```
class CacheSimulator:
    def __init__(self, cache_slots):
        self.cache_slots = cache_slots
        self.cache = circularLinkedListBasic()
        self.cache_hit = 0
        self.tot_cnt = 0
```

CacheSimulator 클래스의 생성자 __init__이다. 리스트를 이용한 방식과 다른 점은 self.cache에 circularLinkedListBasic() 속성을 넣는 것이다.

Simulation 실행 메서드 do_sim(page)

```
    def do_sim(self, page):
        self.tot_cnt += 1
        if self.cache.index(page) != (-2):
            self.cache.remove(page)
            self.cache_hit += 1
        if self.cache.size() >= self.cache_slots:
            self.cache.pop(0)
        self.cache.append(page)
```

이 메서드는 `page`를 참조하는 행동이 발생하였을 때를 시뮬레이션하는 메서드이다.

`self.cache`에 `page`가 있다면 `cache_hit`을 1 증가시킴과 동시에 `self.cache`에서 `page`를 삭제한다. `self.cache`의 크기가 `self.cache_slots` 이상이라면, 처음 원소를 pop하고, page를 리스트의 마지막에 append한다.

결과 출력 메서드 `print_stats()`

```
    def print_stats(self):
        print("cache_slot = ", self.cache_slots, "cache_hit = ", self.cache_hit,
"hit ratio = ", self.cache_hit / self.tot_cnt)
```

이 메서드는 캐시 슬롯의 크기(`self.cache_slots`), 캐시 저장 공간에 접근한 횟수(`self.cache_hit`), 총 시도 중에서 캐시 저장 공간에 접근한 비율(`(self.cache_hit)/(self.tot_cnt)`)을 차례로 출력한다.

출력 결과

Linked List를 활용하여 LRU Simulator를 작동시킨 결과이다.

```
$ python3 lru_sim.py
cache_slot =  100 cache_hit =  14554 hit ratio =  0.14553854461455384
cache_slot =  200 cache_hit =  15377 hit ratio =  0.15376846231537686
cache_slot =  300 cache_hit =  16001 hit ratio =  0.16000839991600083
cache_slot =  400 cache_hit =  16665 hit ratio =  0.16664833351666483
cache_slot =  500 cache_hit =  17628 hit ratio =  0.17627823721762784
cache_slot =  600 cache_hit =  18796 hit ratio =  0.1879581204187958
cache_slot =  700 cache_hit =  20387 hit ratio =  0.2038679613203868
cache_slot =  800 cache_hit =  23945 hit ratio =  0.23944760552394476
cache_slot =  900 cache_hit =  26340 hit ratio =  0.26339736602633973
cache_slot =  1000 cache_hit =  28110 hit ratio =  0.2810971890281097
```

순환 연결 리스트(Circular Linked List) 객체의 구현

Python으로 순환 연결 리스트를 구현한 코드를 보자. 코드는 LRU Simulator에서 사용하는 메서드만을 발췌하였다.

```
class circularLinkedListBasic:
    def __init__(self):
        self.__tail = ListNode('dummy', None)
        self.__numItems = 0
```

생성자 `__init__()`

`self.__tail`은 리스트의 마지막 원소를 Next로 두는 노드이다.

`self.__numItems`는 리스트의 항목 갯수를 저장하는 변수이다.

append 메서드 `append(newItem)`

이 메서드는 리스트의 맨 끝에 `newItem`을 추가하는 메서드이다.

```
    def append(self, newItem):
    newNode = ListNode(newItem, None)
    if self.isEmpty():
        newNode.next = newNode
        self.__tail.next = newNode
        self.__numItems += 1
        return
    else:
        prev = self.__tail.next
        curr = prev.next
        prev.next = newNode
        newNode.next = curr
        self.__tail.next = newNode
        self.__numItems += 1
```

1. newItem을 item으로, None을 nextNode로 갖는 새로운 Node를 만들어 newNode에 저장한다.
2. 만약 리스트가 비었다면
    1. newNode의 다음을 newNode로 하여 순환적으로 만든다.
    2. `self.__tail`의 next 속성을 newNode로 설정하여 newNode를 가리키도록 한다.
    3. `self.__numItems`를 1 증가시킨다.
    4. 메서드 종료
3. 그렇지 않다면
    1. `prev`를 리스트의 마지막 원소(즉, `self.__tail`의 next 속성)으로 정의한다.
    2. `curr`을 리스트의 첫 원소(즉, `self.__tail`의 next의 next 속성)으로 정의한다.
    3. `prev`의 next 속성을 `newNode`로 변경한다.
    4. `newNode`의 next 속성을 `curr`로 변경한다.
    5. `self.__numItems`를 1 증가시킨다.
    6. 메서드 종료

pop 메서드 `self.pop()`

이 메서드는 인자로 넣은 인덱스의 값을 리스트에서 제거하고, 그 값을 리턴하는 함수이다. 만약 인자가 없을 경우 마지막 값을 제거하고 이를 리턴한다.

이 메서드는 조금 복잡하므로, 부분을 나눠서 설명하겠다.

```
    def pop(self, *args):
    if self.isEmpty():
        return None
    if len(args) != 0:
        i = args[0]
    if len(args) == 0 or i == -1:
        i = self.__numItems - 1
```

이 부분은 리스트를 검사하고, 인자 값에 따라 지울 node의 index를 설정하는 부분이다. 리스트가 비었다면, None을 리턴한다. 인자 값이 존재한다면(길이가 0이 아니라면) 지울 node의 index(i)를 첫번째 인수로 설정한다. 인자 값이 존재하지 않거나, 지울 node의 index가 -1이라면, i는 마지막 노드의 index로 설정한다.

```
    if self.size() == 1 and i == 0:
        rtnItem = self.__tail.next.item
        self.__tail.next = None
        self.__numItems -= 1
        return rtnItem
```

리스트의 원소의 갯수가 1이고 지울 node의 index가 0일 경우이다.
return할 item(`rtnItem`)을 마지막 node(`self.__tail`)의 item으로 설정하고, `self.__numItem`을 1 감소시킨다. 이후 `rtnItem`을 return한다.

```
    if i==0:
        prev = self.__tail.next
        curr = prev.next
        rtnItem = curr.item
        prev.next = curr.next
        self.__numItems -=1
        return rtnItem
```

지울 리스트의 index가 0개일 때, prev를 마지막 노드로 설정한다. `prev`의 next를 `curr`로 설정하고, 리턴할 아이템(`rtnItem`)을 `curr`의 item으로 설정한다. 이후 `prev`의 next를 `curr`의 next로 설정한다. `__numItems`에서 1을 빼고, `rtnItem`을 return한다.

```
    if (i >= 1 and i<= self.__numItems - 1):
        prev = self.__getNode(i-1)
        curr = prev.next
        rtnItem = curr.item
        prev.next = curr.next
        self.__numItems -= 1
```

```
        return rtnItem
```

i가 1보다 크거나 같이고 (리스트의 항목 수 - 1)보다 작거나 같은 경우이다. `self.__getNode()` 메서드를 통해 i-1번째 노드를 구하여 `prev`로 설정한다. `curr`을 `prev`의 다음 노드로 설정한다. 리턴할 아이템(`rtnItem`)을 `curr.item`으로 설정하고, `prev.next`를 `curr.next`로 설정한다. `__numItems`에서 `의 값에서 1을 뺀다.

remove 메서드 `remove(x)`

리스트 내에서 특정한 값 `x`를 찾는다. `x`가 있을 경우 이를 리스트에서 제거한 후 `x`를 리턴하고, 없을 경우 `-2`을 리턴한다.

```
    def remove(self, x):
        if self.isEmpty(): return None
        if self.size() == 1:
            if self.__tail.next.item==x:
                self.__numItems -= 1
                self.__tail.next = None
                return x
            else:
                return None
        (prev, curr) = self.__findNode(x)
        if prev==None:
            return None
        prev.next = curr.next
        self.__numItems -= 1
        return x
```

index 메서드 `index(x)`

이 값은 리스트 내에서 특정한 값 `x`의 제로베이스 인덱스 번호를 리턴한다. `x`가 없을 경우 `-2`를 리턴한다.

```
    def index(self, x) -> int:
    if self.isEmpty(): return -2
    curr = self.__tail.next
    for i in range(self.__numItems):
        curr = curr.next
        if curr.item==x:
            return i
    return -2
```

C를 통한 LRU Simulator의 구현

파이썬으로 구현한 코드를 C로 옮기기 위하여, C의 구조체와 함수를 사용하였다. C의 구조체를 객체의 속성을 저장하기 위한 변수로, C의 함수를 메서드로 사용하였다. 이때 혼동을 피하기 위하여 메서드의 이름은 (Python의 메서드의 이름)(객체의 이름)을 카멜 표기법으로 표기하여 적용하였다.

파이썬과 C의 가장 큰 차이는, 메모리 해제이다. 파이썬은 주기적으로 사용하지 않는 객체를 정리하여 메모리 공간을 확보하지만, C는 free()를 통하여 메모리 공간을 돌려주지 않으면 안되기 때문에 destroy 함수를 만들었다. 그리고, Node의 변화가 있을 때마다 그 노드의 메모리 할당을 해제하였다.

배열(Array)을 통한 구현

cachesimulator.h

```
#include <stdio.h>
#include <stdlib.h>

typedef struct cachesimulator {
    int cache_slots;
    int* cache;
    int cache_size;
    int cache_hit;
    int tot_cnt;

} CacheSimulator_t;

CacheSimulator_t* CacheSimulator(int cache_slots_);
void destroyCacheSimulator(CacheSimulator_t* sim);
void do_sim_CacheSimulator(CacheSimulator_t* sim, int page);
void print_stats_CacheSimulator(CacheSimulator_t* sim);
```

cachesimulator.c

```
#include "cachesimulator.h"

CacheSimulator_t* CacheSimulator(int cache_slots_)
{
    CacheSimulator_t* sim = (CacheSimulator_t*)malloc(sizeof(CacheSimulator_t));
    if (!sim) {
        printf("Faled to Alloc\n");
        return NULL;
    }

    sim->cache = (int*)malloc(sizeof(int) * cache_slots_);
    if (!(sim->cache)) {
        printf("Failed to Alloc\n");
        return NULL;
    }
    sim->cache_slots = cache_slots_;
    sim->cache_size = 0;
    sim->cache_hit = 0;
    sim->tot_cnt = 0;

    return sim;
}
```

```
void destroyCacheSimulator(CacheSimulator_t* sim)
{
    free(sim);
    return;
}

void do_sim_CacheSimulator(CacheSimulator_t* sim, int page)
{
    (sim->tot_cnt++);
    int temp = sim->cache_size - 1;
    int temp2 = 0;
    while(temp >= 0)
    {
        if (sim->cache[temp] == page)
        {
            temp2 = 1;
            sim->cache_hit++;
            break;
        }
        temp--;
    }
    if (!temp2)
    {
        if (sim->cache_size == sim->cache_slots)
        {
            for (int i = 0; i < sim->cache_size - 1; i++)
            {
                sim->cache[i] = sim->cache[i + 1];
            }
            sim->cache[sim->cache_size - 1] = page;
        }
        else {
            sim->cache[sim->cache_size] = page;
            sim->cache_size++;
        }
        return;
    }

    while (temp < sim->cache_size-1)
    {
        sim->cache[temp] = sim->cache[temp + 1];
        temp++;
    }
    sim->cache[sim->cache_size - 1] = page;
}

void print_stats_CacheSimulator(CacheSimulator_t* sim)
{
    printf("cache_slot = %d cache_hit = %d hit ratio = %f\n", sim->cache_slots,
sim->cache_hit, (float)sim->cache_hit / (float)sim->tot_cnt);
    return;
}
```

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "cachesimulator.h"

int main(void)
{
    FILE* fp;
    fp = fopen("linkbench.trc", "r");
    int input[100000];
    for (int i = 0; i < 100000; i++)
    {
        fscanf(fp, "%d", &input[i]);
    }

    for (int cache_slots = 100; cache_slots < 1001; cache_slots += 100)
    {
        CacheSimulator_t* cache_sim = CacheSimulator(cache_slots);
        for (int i = 0; i < 100000; i++)
        {
            do_sim_CacheSimulator(cache_sim, input[i]);
        }
        print_stats_CacheSimulator(cache_sim);
        destroyCacheSimulator(cache_sim);
    }
}
```

실행 결과

```
$ ./cachesimulator.exe
cache_slot = 100 cache_hit = 14554 hit ratio = 0.145540
cache_slot = 200 cache_hit = 15377 hit ratio = 0.153770
cache_slot = 300 cache_hit = 16001 hit ratio = 0.160010
cache_slot = 400 cache_hit = 16665 hit ratio = 0.166650
cache_slot = 500 cache_hit = 17628 hit ratio = 0.176280
cache_slot = 600 cache_hit = 18796 hit ratio = 0.187960
cache_slot = 700 cache_hit = 20387 hit ratio = 0.203870
cache_slot = 800 cache_hit = 23947 hit ratio = 0.239470
cache_slot = 900 cache_hit = 26340 hit ratio = 0.263400
cache_slot = 1000 cache_hit = 28110 hit ratio = 0.281100
```

순환 연결 리스트(Circular Linked List)를 통한 구현

cachesimulator.c

```c
#include "cachesimulator.h"

CacheSimulator_t* CacheSimulator(int cache_slots_)
{
    CacheSimulator_t* sim = (CacheSimulator_t*)malloc(sizeof(CacheSimulator_t));
    if (!sim) {
        printf("Faled to Alloc\n");
        return NULL;
    }

    sim->cache = LinkedList();
    sim->cache_size = 0;
    sim->cache_slots = cache_slots_;
    sim->cache_hit = 0;
    sim->tot_cnt = 0;

    return sim;
}

void destroyCacheSimulator(CacheSimulator_t* sim)
{
    free(sim);
    return;
}

void do_sim_CacheSimulator(CacheSimulator_t* sim, int page)
{
    (sim->tot_cnt++);
    if (indexLinkedList(sim->cache, page) != (-2))
    {
        removeLinkedList(sim->cache, page);
        (sim->cache_hit)++;
        appendLinkedList(sim->cache, page);
        return;
    }

    if (sim->cache_size >= sim->cache_slots)
    {
        popLinkedList(sim->cache, 0);
        appendLinkedList(sim->cache, page);
        return;
    }
    else
    {
        (sim->cache_size)++;
        appendLinkedList(sim->cache, page);
        return;
    }

}

void print_stats_CacheSimulator(CacheSimulator_t* sim)
{
```

```
    printf("cache_slot = %d cache_hit = %d hit ratio = %f\n", sim->cache_slots,
sim->cache_hit, (float)sim->cache_hit / (float)sim->tot_cnt);
    return;
}
```

cachesimulator.h

```
#include <stdio.h>
#include <stdlib.h>
#include "Circularinkedlist.h"

typedef struct cachesimulator {
    int cache_slots;
    LinkedList_t* cache;
    int cache_size;
    int cache_hit;
    int tot_cnt;

} CacheSimulator_t;

CacheSimulator_t* CacheSimulator(int cache_slots_);
void destroyCacheSimulator(CacheSimulator_t* sim);
void do_sim_CacheSimulator(CacheSimulator_t* sim, int page);
void print_stats_CacheSimulator(CacheSimulator_t* sim);
```

main.c

```
#pragma warning(disable :4996)
#include <stdio.h>
#include <stdlib.h>
#include "cachesimulator.h"

int main(void)
{
    FILE* fp;
    fp = fopen("linkbench.trc", "r");
    int input[100000];
    for (int i = 0; i < 100000; i++)
    {
        fscanf(fp, "%d", &input[i]);
    }

    for (int cache_slots = 100; cache_slots < 1001; cache_slots += 100)
    {
        CacheSimulator_t* cache_sim = CacheSimulator(cache_slots);
        for (int i = 0; i < 100000; i++)
        {
            do_sim_CacheSimulator(cache_sim, input[i]);
        }
```

```
        print_stats_CacheSimulator(cache_sim);
        destroyCacheSimulator(cache_sim);
    }
}
```

순환 연결 리스트(Circular Linked List)의 구현

circularLinkedlist.h

```
#include <stdio.h>
#include <stdlib.h>

typedef struct ListNodeStruct ListNode_t;
typedef struct LinkedListStruct LinkedList_t;

ListNode_t* ListNode(int item, ListNode_t* next); // ListNode ���� ?L?
LinkedList_t* LinkedList(); //LinkedList ���� ?L?
void destroyListNode(ListNode_t* node);
void destroyLinkedList(LinkedList_t* list);
void insertLinkedList(LinkedList_t* list, int i, int newItem);
void appendLinkedList(LinkedList_t* list, int newItem);
int popLinkedList(LinkedList_t* list, int i);
int removeLinkedList(LinkedList_t* list, int x);
int getLinkedList(LinkedList_t* list, int i);
int indexLinkedList(LinkedList_t* list, int x);
void clearLinkedList(LinkedList_t* list);
int countLinkedList(LinkedList_t* list, int x);
void extendLinkedList(LinkedList_t* list, LinkedList_t* a);
void printLinkedList(LinkedList_t* list);
ListNode_t* findPrevNodeLinkedList(LinkedList_t* list, int x);
ListNode_t* getNodeLinkedList(LinkedList_t* list, int order);
```

circularLinkedList.c

```
#include "linkedlist.h"

typedef struct ListNodeStruct
{
    int item;
    ListNode_t* next;
} ListNode_t;

typedef struct LinkedListStruct
{
    ListNode_t* tail;
    int numItems;
} LinkedList_t;
```

```
ListNode_t* ListNode(int item, ListNode_t* next)
{
    ListNode_t* node = (ListNode_t*)malloc(sizeof(ListNode_t));
    if (!node) {
        printf("Failed to alloc\n");
        return NULL;
    }
    node->item = item;
    node->next = next;

    return node;
}

void destroyListNode(ListNode_t* node)
{
    free(node);
    return;
}

LinkedList_t* LinkedList()
{
    LinkedList_t* tmp = (LinkedList_t*)malloc(sizeof(LinkedList_t));
    tmp->tail = ListNode(-1, NULL);
    tmp->numItems = 0;

    return tmp;
}


void destroyLinkedList(LinkedList_t* list)
{
    clearLinkedList(list);
    destroyListNode(list->tail);
    free(list);
}

void insertLinkedList(LinkedList_t* list, int i, int newItem)
{
    if (i == list->numItems)
    {
        appendLinkedList(list, newItem);
        return;
    }
    else if (i == 0)
    {
        ListNode_t* prev = list->tail->next;
        ListNode_t* curr = prev->next;
        ListNode_t* newNode = ListNode(newItem, NULL);
        prev->next = newNode;
        newNode->next = curr;
        (list->numItems)++;
    }
    else if (i > 0 && i < list->numItems)
    {
```

```
        ListNode_t* prev = getNodeLinkedList(list, i - 1);
        ListNode_t* curr = prev->next;
        ListNode_t* newNode = ListNode(newItem, NULL);
        prev->next = newNode;
        newNode->next = curr;
        (list->numItems)++;
    }
    else
    {
        return;
    }
}

void appendLinkedList(LinkedList_t* list, int newItem)
{
    ListNode_t* newNode = ListNode(newItem, NULL);
    if (list->numItems == 0)
    {
        newNode->next = newNode;
        list->tail->next = newNode;
        (list->numItems)++;
        return;
    }
    else
    {
        ListNode_t* prev = list->tail->next;
        ListNode_t* curr = prev->next;
        prev->next = newNode;
        newNode->next = curr;
        list->tail->next = newNode;
        (list->numItems)++;
        return;
    }
}

int popLinkedList(LinkedList_t* list, int index)
{
    if (index == -1)
        index = list->numItems - 1;
    if (list->numItems == 0)
    {
        return -2;
    }

    if (list->numItems == 1 && index == 0)
    {
        ListNode_t* temp = list->tail->next;
        int rtnItem = temp->item;
        list->tail->next = NULL;
        (list->numItems)--;
        destroyListNode(temp);
        return rtnItem;
    }
```

```
    if (index == 0)
    {
        ListNode_t* prev = list->tail->next;
        ListNode_t* curr = prev->next;
        int rtnItem = curr->item;
        prev->next = curr->next;
        (list->numItems)--;
        destroyListNode(curr);
        return rtnItem;
    }

    if (index >= 1 && index <= list->numItems - 1)
    {
        ListNode_t* prev = getNodeLinkedList(list, index - 1);
        ListNode_t* curr = prev->next;
        int rtnItem = curr->item;
        prev->next = curr->next;
        (list->numItems)--;
        if (index == list->numItems - 1)
            list->tail->next = prev;
        destroyListNode(curr);
        return rtnItem;
    }

    return -2;
}

int removeLinkedList(LinkedList_t* list, int x)
{
    if (list->numItems == 0) return -1;
    if (list->numItems == 1)
    {
        if (list->tail->next->item == x)
        {
            ListNode_t* temp = list->tail->next;
            (list->numItems)--;
            list->tail->next = NULL;
            destroyListNode(temp);
            return x;
        }
        else
        {
            return -1;
        }
    }
    ListNode_t* prev = findPrevNodeLinkedList(list, x);
    if (prev == NULL)
    {
        return -1;
    }
    ListNode_t* curr = prev->next;
    prev->next = curr->next;
    (list->numItems)--;
    destroyListNode(curr);
```

```
    return x;
}

int getLinkedList(LinkedList_t* list, int i)
{
    if (list->numItems == 0)
    {
        return -1;
    }
    else if (i >= 0 && i < list->numItems) {
        ListNode_t* temp = getNodeLinkedList(list, i);
        return (temp->item);
    }
    else {
        return -1;
    }
}

int indexLinkedList(LinkedList_t* list, int x)
{
    if (list->numItems == 0) return -2;
    ListNode_t* curr = list->tail->next;
    for (int i = 0; i < list->numItems - 1; i++)
    {
        curr = curr->next;
        if (curr->item == x) {
            return i;
        }
    }
    return -2;
}

void clearLinkedList(LinkedList_t* list)
{
    if (list->numItems == 0) { return; }
    ListNode_t* temp;
    int i = list->numItems;
    while (i--)
    {
        temp = getNodeLinkedList(list, i);
        destroyListNode(temp);
    }
    list->tail->next = NULL;
    list->numItems = 0;
}

ListNode_t* findPrevNodeLinkedList(LinkedList_t* list, int x)
{
    if (list->numItems == 0) return NULL;
    ListNode_t* prev = list->tail->next;
    ListNode_t* curr = prev->next;
    while (curr != list->tail->next)
    {
        if (curr->item == x)
```

```
            {
                return prev;
            }
            prev = prev->next;
            curr = curr->next;
        }

        return NULL;
    }

    ListNode_t* getNodeLinkedList(LinkedList_t* list, int order)
    {
        if (list->numItems == 0) return NULL;
        if (order == list->numItems - 1 || order == -1) return list->tail->next;
        if (order >= list->numItems) {
            return NULL;
        }
        ListNode_t* curr = (list->tail->next->next);

        for (int i = 0; i < order; i++)
        {
            curr = curr->next;
        }
        return curr;
    }
```

실행 결과

```
$ ./main.exe
cache_slot = 100 cache_hit = 14532 hit ratio = 0.145320
cache_slot = 200 cache_hit = 15355 hit ratio = 0.153550
cache_slot = 300 cache_hit = 15979 hit ratio = 0.159790
cache_slot = 400 cache_hit = 16643 hit ratio = 0.166430
cache_slot = 500 cache_hit = 17606 hit ratio = 0.176060
cache_slot = 600 cache_hit = 18774 hit ratio = 0.187740
cache_slot = 700 cache_hit = 20365 hit ratio = 0.203650
cache_slot = 800 cache_hit = 23923 hit ratio = 0.239230
cache_slot = 900 cache_hit = 26318 hit ratio = 0.263180
cache_slot = 1000 cache_hit = 28088 hit ratio = 0.280880
```

제언

실행 속도를 비교해 보니, Python보다 C로 작성한 코드가 월등히 빨랐다. 각 Array와 CircularLinkedList를 통해 작성한 코드를 비교하면 C의 경우 실행 시간은 비슷했지만, Python의 경우 CircularLinkedList가 현저히 느렸다. 이는 CircularLinkedList의 경우 알고리즘 최적화의 미비로 리스트 순회 횟수가 너무 많아 파이썬에서 이 차이가 뚜렷히 보였음으로 예상할 수 있다.