2020년도 2학기 컴퓨터공학설계및실험Ⅰ

8주차 테트리스 2주차 예비보고서

20161663 허재성

1. 실습 목적

일반적인 게임은 사용자의 기록을 등록하여, 확인할 수 있는 기능을 갖는다. 테트리스 프로젝트에서도 이러한 기능을 하는 랭킹 시스템을 구현한다. 이 시스템은 테트리스 플레이가 종료되면, 사용자의 이름과 점수를 기록하고, 기록된 랭킹 정보들을 확인할 수 있는 기능을 갖는다. 이 랭킹 정보들은 테트리스 프로그램 내에서 효율적인 자료구조를 사용하여 관리되고, 테트리스 프로그램을 종료할 때는 rank.txt 파일에 기록되어 지속적으로 유지된다.

2. 관련 이론

연결 리스트(Linked List)에 대해 조사한 내용과, 다른 자료구조에 비해 어떤 장, 단점을 가지는지 서술하시오.

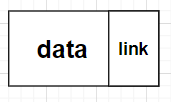
리스트(List)는 컴퓨터 과학에서 중복된 값이 한 번 이상 존재할 수 있는 일련의 값이 모여 있는 추상적 자료형으로 시퀀스(sequence)라고도 부른다. 리스트 자료형은 대표적으로 배열(Array) 자료구조 또는 연결 리스트(Linked List) 자료구조로 구현될 수 있다.[1]

연결 리스트(Linked List)는 추상적 자료형인 리스트를 구현한 자료 구조로 각 노드가 데이터와 포인터를 가지고 한 줄로 연결되어 있는 방식으로 데이터를 저장하는 자료구조이다. 데이터를 담고 있는 노드들이 연결되어 있는데, 노드의 포인터가 다음이나 이전의 노드와의 연결을 가지고 있다.[2]

다음 코드는 연결 리스트의 노드를 구현한 예이다. 구조체 이름을 Node라 지었다.

|  |
| --- |
| typedef struct \_Node {  int data;  /\* other fields \*/  struct \_Node\* link;  } Node; |

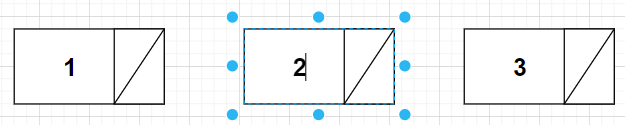
위의 Node 구조체는 데이터로 정수형 변수 data를 가지고 있다(필요시 다른 변수를 추가할 수 있다.). Node 구조체에는 구성 요소에 데이터 필드인 data 외에도, 자기 자신(Node)을 가리킬 수 있는 포인터 변수 link가 존재한다. 이처럼 구조체의 구성 요소 중에 자기 자신을 가리키는 포인터가 1개 이상 존재하는 구조를 자기 참조 구조(self-referential structure)라고 한다. 자기 참조 구조는 명시적으로 메모리를 할당받고 반납받기 위해, 통상적으로 동적 저장 공간 관리 루틴(malloc, free)을 필요로 한다.[3] 연결 리스트의 노드는 자기 참조 구조로 구현할 수 있다.



구현한 노드 구조체인 Node를 그림으로 나타낸 것이다.

|  |
| --- |
| Node\* item1, \*item2, \*item3;  item1 = (Node\*)malloc(sizeof(Node));  item2 = (Node\*)malloc(sizeof(Node));  item3 = (Node\*)malloc(sizeof(Node));  item1->data = 1;  item2->data = 2;  item3->data = 3;  item1->link = item2->link = item3->link = NULL; |

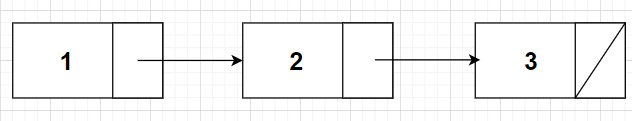
Node 구조체의 포인터 변수 item1, item2, item3를 선언하고 malloc으로 Node 구조체를 동적할당하여 각각 item1, item2, item3가 가리키도록 한다. item1, item2, item3의 데이터 data를 각각 1, 2, 3으로 초기화하고, item1, item2, item3의 link를 NULL로 초기화한다. 아직 노드끼리 서로 연결되지 않았다는 의미이다.



위의 그림과 같이 각 노드들의 데이터는 1, 2, 3으로 초기화했고, link는 모두 NULL로 초기화하여 각 노드들이 서로 연결되지 않게 하였다. link가 NULL일 경우 위의 그림과 같이 대각선으로 표시한다.

|  |
| --- |
| item1->link = item2;  item2->link = item3; |

item1 노드의 link가 item2 노드를 가리키게 하고, item2 노드의 link가 item3를 가리키게 한다. 다음 그림과 같이 된다..

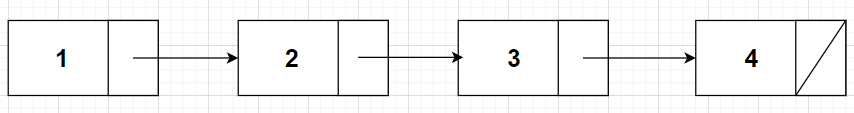


item1에 item2가 연결되고, item2에 item3가 연결되었다. 각 노드가 서로 연결되어 연결 리스트(Linked List)가 구현되었다. 위에서는 item1이 연결 리스트의 첫 번째 노드(head)가 되고, item3이 연결 리스트의 마지막 노드가 된다.

추상 자료형 리스트에서는 기본적으로 리스트의 특정 위치의 노드에 접근, 리스트의 특정 위치에 새로운 노드 삽입, 기존의 리스트에서 특정 노드 삭제, 이렇게 세 가지 연산을 생각할 수 있다. 연결 리스트에서 이 세 가지 연산을 구현할 수 있다.

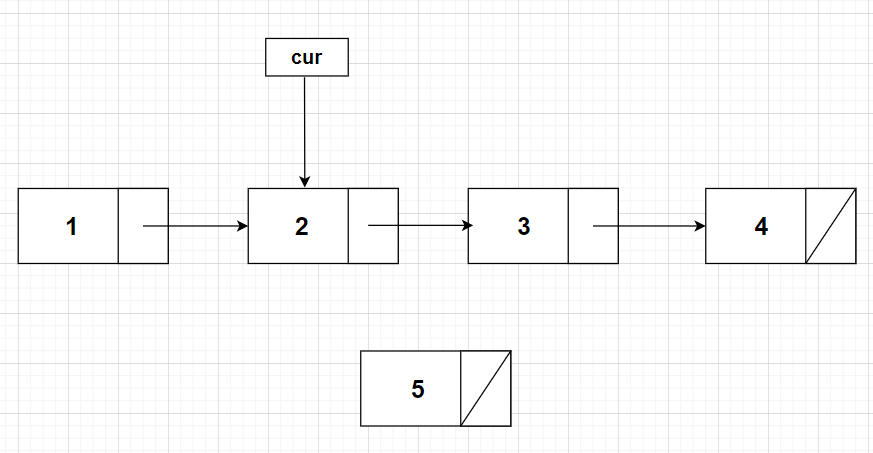
특정 위치의 노드에 접근

특정 위치의 노드에 접근하는 연산이다. 특정 위치란 앞에서부터 3번째, 뒤에서부터 2번째와 같은 위치를 말한다. 배열에서 index와 유사하다. 연결 리스트의 노드의 개수가 N개일 때, 특정 위치 i(i=1~N)에 접근하기 위해서는 연결 리스트의 head 노드부터 시작해서 link를 따라서 총 i번 탐색해야 i번째 리스트에 접근할 수 있다. 최악의 경우(i=N일 경우) N번 탐색해야 한다. 따라서 특정 위치에 접근하는 연산의 시간 복잡도는 O(N)이다.



위의 그림에서 첫 번째 노드인 item1에 접근하는 것은 한 번에 바로 접근이 가능하지만, data=4인 네 번째 노드 item4에 접근하려면 item1부터 시작해서 link를 따라 item2, item3를 접근한 뒤에 접근이 가능하다. 즉 4번의 접근이 필요하다. 기본적으로 구현된 단일 연결 리스트(singly linked list)에서는 특정 노드에 바로 접근할 수 없다. 최대 O(N)의 시간이 걸린다.

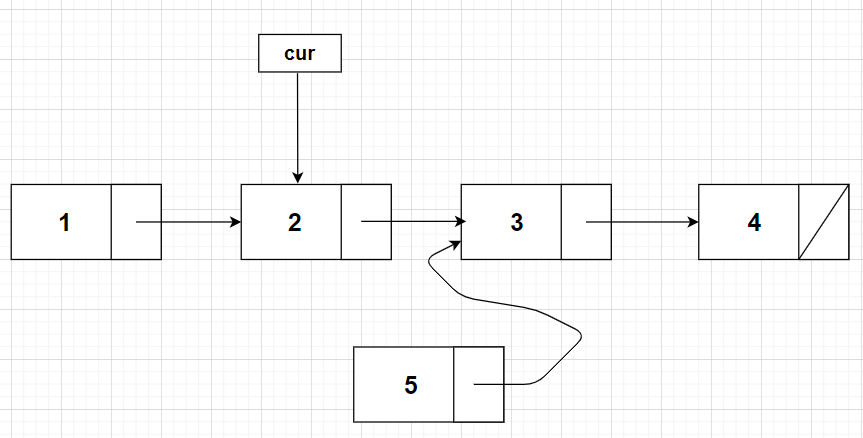
특정 위치에 새로운 노드 삽입



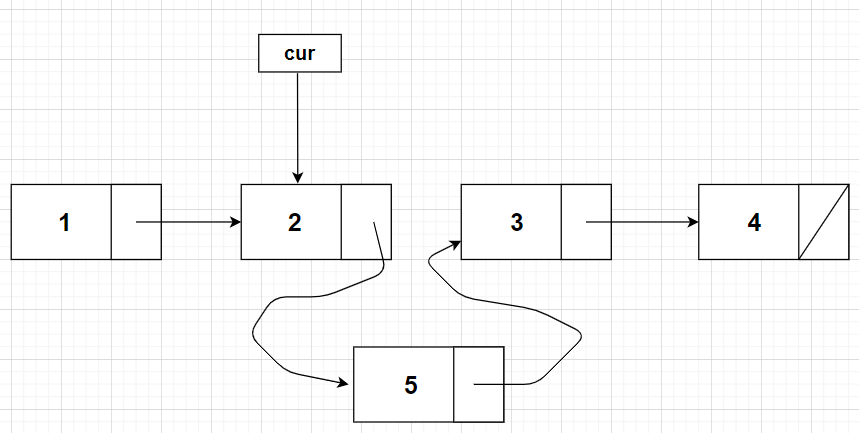
위와 같이 노드 item2와 item3 사이에 data=5인 item5 노드를 삽입한다고 가정한다. 이 때 삽입할 위치인 item2(정확히는 item2 뒤에 삽입)를 가리키는 포인터(cur)이 있다고 가정한다. 삽입할 위치를 가리키는 포인터가 있으면 삽입은 O(1) 시간에 이루어진다.

|  |
| --- |
| item5->link = cur->link;  cur->link = item5; |

item5의 link를 cur 노드(item2 노드)의 link가 가리키는 노드인 item3를 가리키게 한다.(첫 번째 문장)

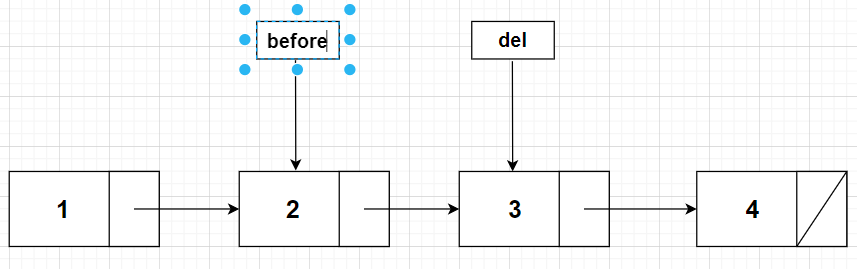


첫 번째 문장이 수행된 뒤의 상황은 위와 같다. O(1)의 시간이 소요된다.



두 번째 문장이 수행되고 삽입이 완료된 모습이다. 두 번째 문장도 O(1)의 시간이 소요되어 삽입에 소요되는 전체 시간의 시간 복잡도도 O(1)이다. 연결 리스트에서의 삽입은 O(1) 시간에 이루어진다.

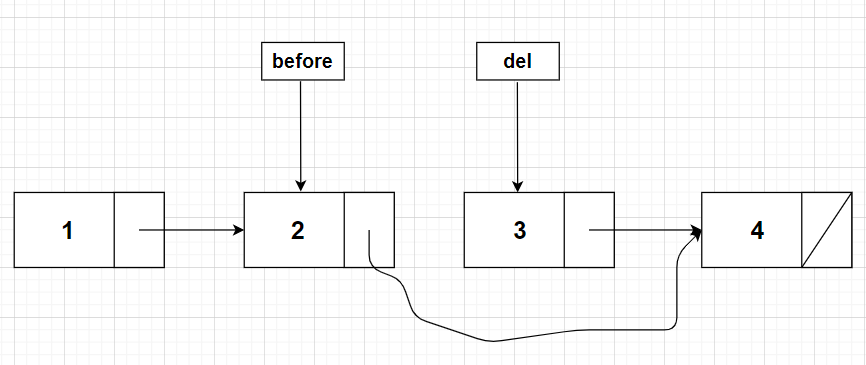
기존의 리스트에서 특정 노드 삭제



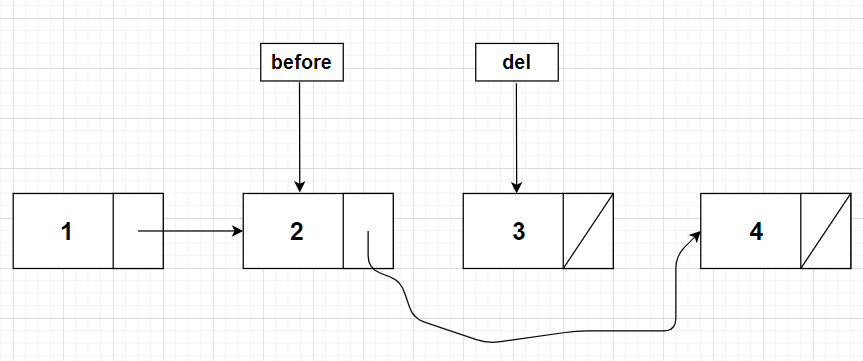
위의 연결 리스트에서 item3을 삭제한다고 가정한다. 삭제할 노드를 del 포인터로 가리키고, 삭제할 노드의 이전 노드를 before 포인터가 가리킨다고 가정하면 del 노드의 삭제는 O(1) 시간에 이루어진다.

|  |
| --- |
| before->link = del->link;  del->link = NULL;  free(del); |

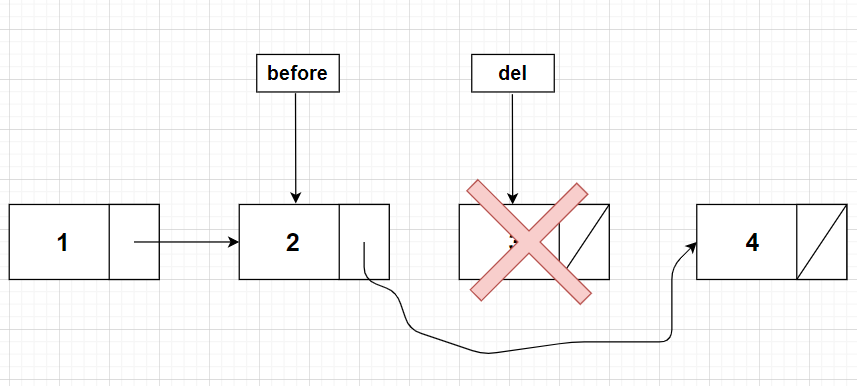
첫 번째 문장에서 before 노드의 link가 삭제하려는 del 노드의 link가 가리키는 노드, 즉 del 노드의 다음 노드를 가리키도록 한다. 첫 번째 문장이 수행된 뒤의 모습은 다음과 같다.



두 번째 문장에서 del 노드의 link를 NULL로 한다. 이 과정은 생략 가능하지만 의미를 명확하게 하기 위해서 수행하는 것이 좋다.



두 번째 문장까지의 수행 만으로도 item3은 연결 리스트로부터 분리되었고(삭제 되었고) 삭제 후에도 item2와 item4가 연결되어 연결 리스트 구조가 유지됨을 알 수 있다. 하지만 item3는 동적 할당하였기 때문에 메모리 누수를 막기 위해 free로 메모리를 해제해줘야 한다.



세 번째 문장을 수행하면 del 노드가 가리키는 동적 할당된 메모리가 해제된다. 각 문장의 수행 시간은 O(1)이므로 리스트에서 특정 노드를 삭제하는 연산의 시간 복잡도는 O(1)이다.

연결 리스트 자료구조의 장단점을 리스트 추상 자료형을 구현하는 또 다른 자료구조인 배열 자료구조와 비교하며 알아본다.

특정 노드(인덱스)에 접근하려는 연산을 비교해보면 다음과 같다.

배열 : O(1) 시간 소요

배열 arr에서는 특정 인덱스 i에 접근하려면 arr[i]로 접근할 수 있다. i가 배열의 범위(0 <= i <= size -1)일 때 arr[i] = k; 또는 k = arr[i]; 로 O(1) 시간에 접근이 가능하다.

연결 리스트 : O(N) 시간 소요

기본적인 연결 리스트의 경우, 특정 노드에 접근하기 위해서는 head 노드에서부터 시작하여 해당 노드가 나올 때 까지 링크를 따라 접근해야 한다. 해당 노드가 앞 쪽에 있을 수록(head에 가까울 수록) 빠른 접근이 가능하지만, 최악의 경우 접근하려는 노드가 마지막 노드일 경우 접근 시간은 O(N)이다. (N은 연결 리스트의 노드의 개수)

특정 위치에 새로운 노드(원소) 삽입 연산을 비교해보면 다음과 같다.

배열 : O(N) 시간 소요

배열의 경우 특정 인덱스에 새로운 원소를 삽입할 경우 최악의 경우, O(N) 시간이 걸린다. 편의 상 배열의 크기는 충분히 크다고 가정할 때, 기존 배열의 원소의 개수가 N개이면, 배열의 마지막 원소의 인덱스는 N-1이다. 배열의 끝에 새로운 원소를 삽입할 경우 arr[N] = newdata; 로 O(1) 시간에 삽입이 가능하다. 하지만 배열의 중간에 i번째 인덱스에 새로운 원소를 삽입할 경우 기존의 i번째 원소로부터의 모든 원소는 한 칸씩 뒤로 이동해야 한다. 즉 i번째 인덱스에 특정 원소를 삽입하면 기존의 i번째 원소부터 N-1번째 원소까지 모든 원소(총 N-i 개의 원소)가 한 칸씩 뒤로 이동해야 한다. 만약 i가 0일 경우, 즉 첫 번째 인덱스에 삽입할 경우, 기존의 모든 배열의 원소를 이동시켜야 하므로 O(N) 시간이 소요된다.

연결 리스트 : O(1) 시간 소요

연결 리스트의 삽입할 위치를 가리키는 포인터가 있으면 해당 포인터가 가리키는 노드의 뒤에 새로운 노드를 삽입하는데 걸리는 시간은 O(1)이다. 연결 리스트의 크기에 관련 없이 상수 시간에 수행 가능하다.

기존의 리스트에서 특정 노드(원소) 삭제 연산을 비교해보면 다음과 같다.

배열 : O(N) 시간 소요

배열의 경우 특정 인덱스의 원소를 삭제할 경우 최악의 경우, O(N) 시간이 걸린다. 기존 배열의 원소의 개수가 N개이면, 배열의 마지막 원소의 인덱스는 N-1이다. 배열의 끝의 원소를 삭제할 경우 arr[N-1] = -1; 로 O(1) 시간에 삭제가 가능하다.(삭제된 원소를 -1로 가정한다.) 하지만 배열의 중간의 i번째 원소를 삭제할 경우 기존의 i+1번째 원소부터 모두 한 칸씩 앞으로 이동해야 한다. 즉 i번째 인덱스의 특정 원소를 삭제시키면 기존의 i+1번째 원소부터 N-1번째 원소까지 모든 원소(총 N-i-1 개의 원소)가 한 칸씩 앞으로 이동해야 한다. 만약 i가 0일 경우, 즉 첫 번째 인덱스를 삭제할 경우, 삭제 후 남은 기존의 모든 배열의 원소를 이동시켜야 하므로(N-1번 이동), O(N) 시간이 소요된다.

연결 리스트 : O(1) 시간 소요

연결 리스트의 삭제할 노드를 가리키는 포인터와 이전 노드를 가리키는 포인터가 있으면 노드를 삭제하는데 걸리는 시간은 O(1)이다. 연결 리스트의 크기에 관련 없이 상수 시간에 수행 가능하다.

위의 비교 결과를 표로 정리해보면 다음과 같다.

|  |  |  |
| --- | --- | --- |
|  | **배열(Array)** | **연결 리스트(Linked List)** |
| **접근(access)** | **O(1)** | **O(N)** |
| **삽입(insertion)** | **O(N)** | **O(1)** |
| **삭제(delete)** | **O(N)** | **O(1)** |

연결 리스트 자료구조는 배열 자료구조와 비교했을 때, 리스트 내 원소에 접근하는데 시간이 더 오래 걸린다. 배열은 원소의 위치에 관계없이 O(1)의 시간이 걸리고, 연결 리스트는 원소의 위치에 따라 최악의 경우O(N)의 시간이 걸린다. 접근 연산의 경우 연결 리스트가 배열보다 소요 시간에 있어서 불리하다.

연결 리스트 자료구조는 배열 자료구조와 비교했을 때, 리스트 내의 특정 위치에 원소를 삽입하는 시간이 더 적게 걸린다. 배열은 삽입 위치에 따라 최악의 경우 O(N)의 시간이 걸리고, 연결 리스트는 삽입 위치에 관계 없이 O(1)의 시간이 소요된다. 삽입 연산의 경우 연결 리스트가 배열보다 소요 시간에 있어서 유리하다.

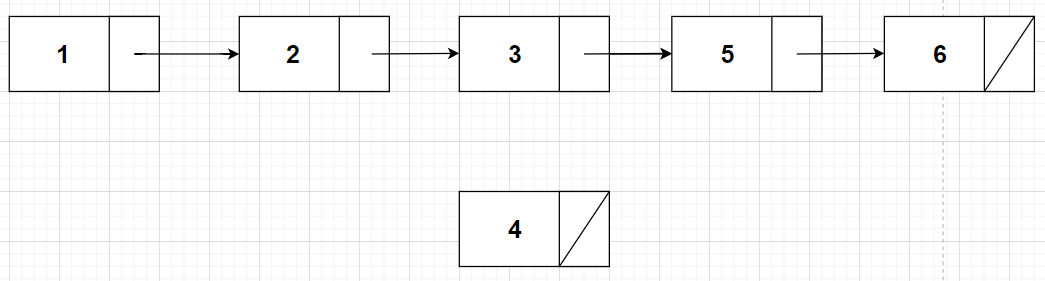
연결 리스트 자료구조는 배열 자료구조와 비교했을 때, 리스트 내의 특정 위치의 원소를 삭제하는 시간이 더 적게 걸린다. 배열은 삭제 위치에 따라 최악의 경우 O(N)의 시간이 걸리지만, 연결 리스트는 삭제 위치에 관계 없이 O(1)의 시간이 소요된다. 삭제 연산의 경우에도 연결 리스트가 배열보다 소요 시간의 관점에서 유리하다.

정리해보면 연결 리스트는 삽입, 삭제 연산에서 기존의 배열보다 유리하고, 접근 연산에서는 기존 배열보다 불리하다. 추가적으로 배열의 경우, 기본 자료구조인 배열을 사용하여 쉽게 사용할 수 있지만, 연결 리스트의 경우 구현이 배열에 비해 까다롭다. 또한 메모리를 동적 할당하고 해제해줘야 하므로 메모리 누수가 발생하지 않도록 주의해야 한다.

3. 실습 방법

연결 리스트(Linked List)에 저장된 데이터를 어떻게 정렬할 지 해결 방법을 서술하시오. 해결 방법을 의사 코드(Pseudo Code)로 작성한 후, 시간 및 공간 복잡도를 계산하시오.

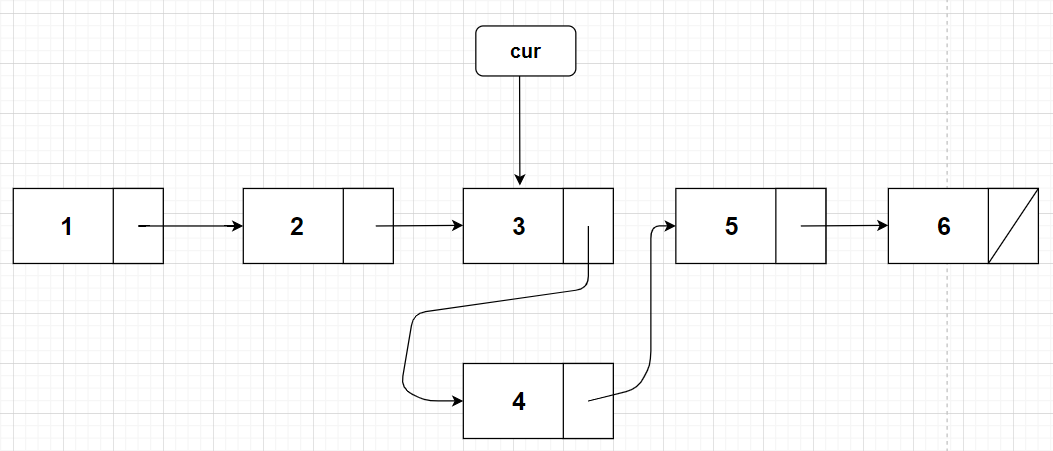
이미 정렬된 연결 리스트에 새로운 원소를 삽입 시, 삽입 이후에도 정렬 상태가 유지되게 하는 방법을 알아본다.



위의 그림에서 이미 비내림차순으로 정렬된 연결리스트 [1, 2, 3, 5, 6]에 데이터가 4인 노드 newNode를 삽입하는 상황을 가정해보면, newNode는 연결 리스트의 노드 3(데이터가 3인 노드) 다음, 정확히 말하면, 노드 3과 노드 5 사이에 삽입되어야 한다. 삽입된 결과는 [1, 2, 3, 4, 5, 6]이 되어야 한다.

newNode의 삽입을 위해서 이미 정렬된 연결 리스트의 head(가장 앞의 노드)부터 관찰한다. 삽입할 위치의 노드를 포인터 cur로 가리키면, newNode를 cur의 뒤에 연결하여 삽입한다. 정렬된 순서를 유지하기 위해 알맞은 cur의 위치를 찾아야 한다. 기존 연결 리스트의 가장 앞의 노드를 head, 새로 삽입할 노드를 newNode라 할 때, 노드를 삽입하는 함수를 insertNode라고 하자. insertNode 함수의 의사 코드는 다음과 같다.

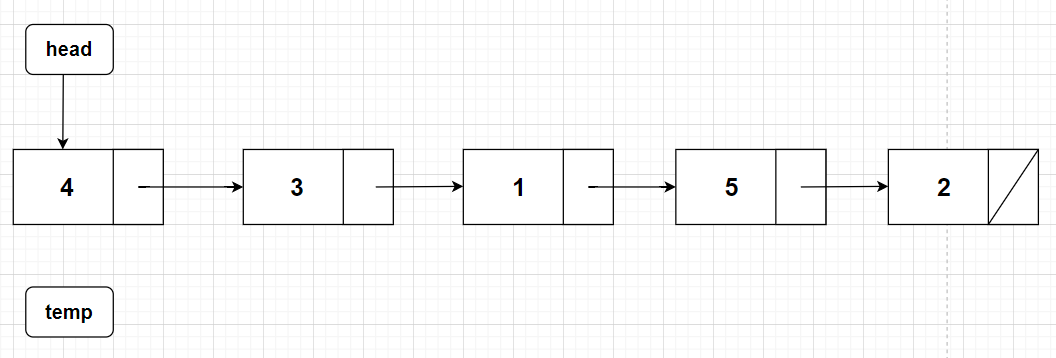
|  |
| --- |
| void insertNode(Node\*\* head, Node\* newNode) {  Node\* cur = \*head;  if (첫 번째 노드가 NULL일 경우) {  \*head = newNode;  return ;  } else if (cur의 data가 newNode의 데이터보다 큰 경우) {  newNode->link = cur; // 첫 번째 노드 앞에 삽입  \*head = newNode; // newNode가 첫 번째 노드가 됨.  return;  } else {  while(다음 노드가 존재할 경우) {  if (삽입 위치를 찾은 경우) {  newNode->link = cur->link;  break;  }  cur = cur->link;  }  cur->link = newNode; // 다음 노드가 존재하지 않을 경우에는 끝에 삽입  }  } |



위의 방법대로 cur이 노드 3을 가리킬 때, 삽입이 이루어진다. newNode의 link가 cur->link(cur의 다음 노드)를 가리키고, cur->link는 newNode가 된다.(cur 노드의 다음 노드가 newNode가 된다.)

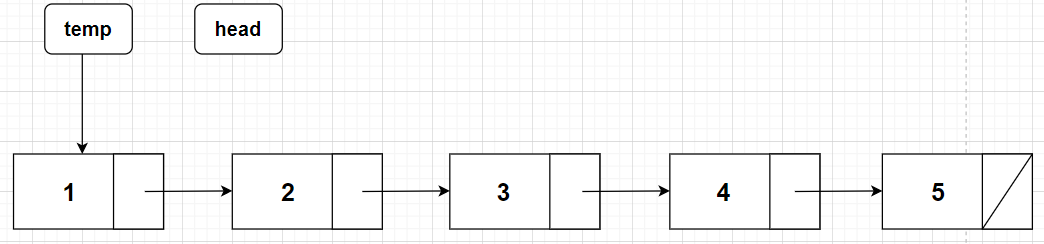
이미 정렬된 연결 리스트에 새로운 노드를 정렬 순서에 맞게 삽입하는 경우 N이 원소의 개수이고, 새로 삽입하려는 노드의 데이터가 연결 리스트의 마지막 노드의 데이터보다 클 때 최대 N번의 비교가 필요하다. 최악의 경우 O(N)의 시간 복잡도를 가지게 된다.

이미 정렬된 연결 리스트에 새로운 노드를 삽입한 뒤에도 정렬 순서를 유지하는 방법을 응용해 무작위 순서로 연결된 리스트를 정렬할 수 있다.

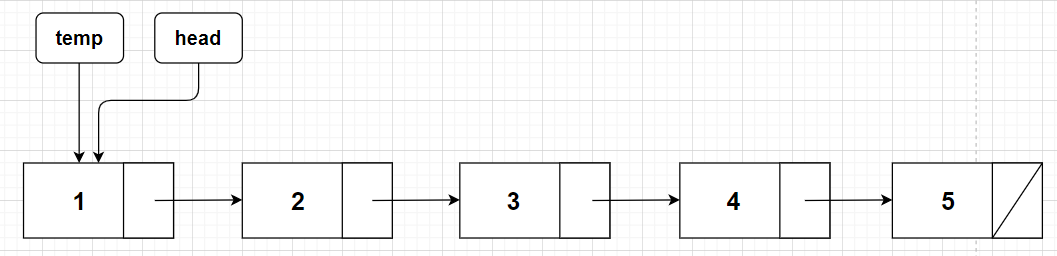


위와 같이 무작위 순서로 연결된 연결 리스트를 정렬하기 위해서 insertNode 함수를 이용할 수 있다.

기존의 무작위 순서로 연결된 리스트를 head라고 하고 임시로 사용할 연결 리스트를 temp라고 하면 head의 가장 앞의 원소부터 하나씩 제거하여 제거된 노드를 연결 리스트 temp에 insertNode 함수를 이용해 삽입하면, temp가 가리키는 연결 리스트는 정렬된 상태가 된다.



마지막으로 포인터 변수 head가 포인터 변수 temp를 가리키게 하면 기존의 연결 리스트 head의 정렬이 완료된다.



다음은 이러한 정렬 과정을 나타낸 의사코드이다.

|  |
| --- |
| void sort(Node\*\* head) {  Node\* cur = \*head;  Node\* temp = NULL;  Node\* del;  \*head = NULL;  while(연결 리스트의 노드가 존재할 경우) {  del = cur;  cur = cur->link;  del->link = NULL; // 연결 리스트에서 가장 앞의 노드를 떼어냄  insertNode(&temp, del); // temp에 뗴어낸 노드를 삽입함.  }  \*head = temp; // head가 정렬된 연결 리스트를 가리킴  } |

위와 같은 정렬 방법은 원소의 개수가 N개일 때 최악의 경우 시간 복잡도가 O(N2)이다. head에서

i번째 원소를 제거하여 temp 원소에 삽입할 때 시간 복잡도는 최대 i-1번의 비교가 필요하다. Head의 원소의 개수가 N개이므로 최대 비교 횟수는 0 + 1 + 2 + … + N – 1 = N(N – 1)/2이다. 각 비교와 삽입의 시간 복잡도는 O(1)이므로 전체 시간 복잡도는 O(1) \* O(N(N-1)/2) = O(N2)이다. N개의 원소를 저장하기 위해 N개의 노드를 할당해야 하므로 공간 복잡도는 O(N)이다. 정렬을 위해 추가적으로 필요한 공간은 없으므로 정렬의 공간 복잡도는 O(1)이다.

4. 기타

4주차 실습에서 학습한 것처럼 연결 리스트 자료구조를 이용하여 스택 자료구조(Stack)와 큐(Queue) 자료 구조를 구현할 수 있다. 위에서 구현한 연결 리스트의 구현 방식을 수정하여 삽입과 삭제를 모두 연결 리스트의 가장 앞에서 이루어지게 하면 스택 자료구조가 구현되고. 삽입을 연결 리스트의 가장 앞에서, 가장 마지막 원소에서 삭제가 이루어지게 하면 큐 자료구조가 구현된다. 스택의 경우, 삽입과 삭제 모두 O(1) 시간에 이루어진다. 큐의 경우, 큐의 가장 앞의 노드를 가리키는 포인터 변수와, 가장 마지막 노드를 가리키는 포인터 변수를 정의하고, 노드의 삽입과 삭제가 이루어질 때마다 포인터 변수가 갱신되도록 유지하면 큐에서도 삽입과 삭제 모두 O(1) 시간에 이루어질 수 있다.

위에서 알아본 단순 연결 리스트(Singly Linked List)를 확장하여, 연결 리스트의 마지막 노드의 링크가 연결 리스트의 첫 번째 노드를 가리키게 하는 원형 연결 리스트(Circularly linked list)를 구현할 수 있다. 또한 연결 리스트의 노드에 기존의 다음 노드를 가리키는 link 필드와 같이 이전 노드를 가리키는 llink(left-link)를 추가하여 연결 리스트의 노드가 이전 노드와 다음 노드를 모두 가리킬 수 있는 이중 연결 리스트(Doubly linked list)를 구현할 수 있다. 이중 연결 리스트를 이용하면 노드를 삭제할 때, before와 같은 포인터 변수를 사용할 필요 없이 O(1) 시간에 삭제할 수 있다.

연결 리스트의 노드와 비슷한 방법으로 트리(Tree) 자료구조의 노드를 구현할 수 있다.

5. 참고 문헌

[1] 위키백과 “리스트(컴퓨팅)”

https://ko.wikipedia.org/wiki/%EB%A6%AC%EC%8A%A4%ED%8A%B8\_(%EC%BB%B4%ED%93%A8%ED%8C%85)

[2] 위키백과 “연결 리스트”

https://ko.wikipedia.org/wiki/%EC%97%B0%EA%B2%B0\_%EB%A6%AC%EC%8A%A4%ED%8A%B8

[3] [Ellis Horowitz](http://www.yes24.com/SearchCorner/Result?domain=ALL&author_yn=Y&query=Ellis+Horowitz), [Sartaj Sahni](http://www.yes24.com/SearchCorner/Result?domain=ALL&author_yn=Y&query=Sartaj+Sahni), [Susan Anderson-Freed](http://www.yes24.com/SearchCorner/Result?domain=ALL&author_yn=Y&query=Susan+Anderson-Freed), “Fundamentals of Data Structures in C”, [*Silicon Valley Publishers Group*](javascript:void(0);), 2007