교본: C언어로 배우는 알고리즘

지은이: 이재규

3.4 연결 리스트(Linked List)

연결 리스트 = 노드(node) + 링크(link)

- node: 실제의 정보를 담고 있는 하나의 단위

- link: 인접 노드의 위치를 저장하고 있어 연결 리스트의 순서를 유지할 수 있게 하는 연결 고리

|  |  |
| --- | --- |
| 배열(array) | 연결 리스트(Linked list) |
| 자료구조가 정적이다. | 자료구조가 동적이다. |
| 여분의 공간을 마련할 필요가 있음  Ex) int arr[20]; // int(4byte) 20개 | 할당, 해제로 메모리 관리가 가능   * 메모리를 절약할 수 있음 |
| 메모리가 연속된 공간을 차지함 | 링크(Link)에 의해 순서를 유지함 |
| 재배열(rearrangement)이 어렵다.  당기고 미는 등의 메모리 복사가 필요 | 재배열(rearrangement)이 쉽다.  링크가 가리키는 방향만 바꿈 |

정리: 연결리스트는 동적으로 메모리를 사용하기 때문에 프로그램의 실행 중 얼마든지 규모를 크게 하든지, 작게 할 수 있다.

- 연결 리스트는 각 노드(node)별로 링크의 개수와 링크의 연결 상태에 따라 단순 연결 리스트, 환형 연결 리스트, 이중 연결 리스트, 이중 환형 연결 리스트 등 이 있음

3.4.1 단순 연결 리스트(Simple Linked List)

단순 연결 리스트 = 정보를 저장하는 노드 + 다음의 노드를 가리키는 링크

- 단순 연결 리스트는 t(노드)의 뒤의 노드만을 알 수 있지 앞의 노드가 무엇인지 알 수 없다.

|  |
| --- |
| \*\*단순 연결 리스트 노드 |
| **struct \_node**  **{**  **int key; // 정보 저장**  **struct \_node \*next; // 다음 노드의 위치 저장**  **} node;** // 태그명(tag name) |

- 포인터는 어떤 형의 데이터를 가리키든지 그 자체의 크기는 변함이 없다.

이유: 데이터형과는 상관없이 단지 주소를 저장하기 때문

|  |
| --- |
| \*\*초기화 함수 |
| **node \*head, \*tail;**  **void init\_list(void)**  **{**  **head = (node\*)malloc(sizeof(name)); // 머리의 공간 확보**  **tail = (node\*)malloc(sizeof(name)); // 꼬리의 공간 확보**  **head->next = tail; // 머리의 다음은 꼬리**  **tail->next = tail; // 꼬리의 다음은 꼬리 ①**  **}** |

① tail이 NULL이나 기타 엉뚱한 곳을 가릴 킬 경우, while문 같은 루프의 내부에서 t = t->next 같은 문장이 있다면 t의 메모리가 이상한 곳에 위치하게 됨으로

이를 방지하기 위해 tail->next = tail을 씀

|  |
| --- |
| \*\*노드를 삽입하는 함수 \*p.130 <그림 3-12> 참고 |
| **node \*insert\_after(int k, node\* t)**  **{**  **node \*s;**  **s = (node\*)malloc(sizeof(node));**  **s->key = k;**  **s->node = t->next;**  **t->next = s;**  **return s;**  **}** |

|  |
| --- |
| \*\*노드를 삭제하는 함수 \*p.131 <그림 3-13> 참고 |
| int delet\_next(node \* t)  {  node \*s;  if (t->next == tail) // t의 다음 노드가 꼬리인가?  Return 0; // 꼬리를 자를 수는 없다.  s = t->next; // 삭제할 노드를 s가 물어둠  t->next = t->next->next; // t의 next는 한 노드를 건너뛴다.  free(s); // 물어둔 노드를 삭제  return 1;  } |

|  |
| --- |
| \*\*할당받은 키(key)를 연결 리스트에서 찾아내는 함수 |
| node \*find\_node(int k)  {  node \*s;  s = head->next; // head->next는 연결 리스트의 처음 노드  while(s->next != k && s != tail) // key를 찾거나 tail에 도달하면 끝  s = s->next; // 다음 노드로 넘어감  return s;  } |

|  |
| --- |
| \*\*인자로 주어진 정수 값을 가진 노드를 찾아서 삭제하는 함수 |
| int delete\_node(int k)  {  node \*s; // 검색을 할 노드  node \*p; // s가 가리키는 노드의 앞 노드  p = head;  s = p->next;  while(s->key != k && s != tail) // 키 값을 찾거나 tail에 도달하면 끝  {  p = p->next; // p는 다음 노드로  s = p->next; // s는 p의 다음 노드로  }  if (s != tail) // s가 tail이 아니면 키 값을 찾은 것이다.  {  p->next = s->next; // p의 다음 노드는 s의 다음 노드  free(s); // 즉 s는 연결에서 빠진다.  return 1;  }  else  return 0;  } |

|  |
| --- |
| \*\*주어진 k(key)값 앞의 노드에 t를 삽입 |
| node \*insert\_node(int t, int k)  {  node \*s; // 키 검색을 따라가는 포인터  node \*p; // s의 앞 노드를 가리키는 포인터  node \*r; // 삽입하는 노드를 만들기 위한 포인터  p = head;  s = p->next;  while(s->key != k && s != tail) // k 키 값을 찾음  {  p = p->next;  s = p->next;  }  if (s != tail) // 찾으면  {  r = (node\*)malloc(sizeof(node)); // r노드 생성  r->key = t; // 노드에 키 값 설정  p->next = r; // p의 다음은 r  r->next = s; // r의 다음은 s, 즉 p -> r -> s  } // r은 p와 s 사이에 삽입됨  return p->next;  } |

|  |
| --- |
| \*\*정렬(sort)된 노드에 순서에 맞게 사이에 삽입하는 함수 |
| node \*ordered\_insert(int k)  {  node \*s; // 검색을 따르는 포인터  node \*p; // s의 앞 노드를 가리키는 포인터  node \*r; // p와 s 사이에 삽입될 노드의 포인터  p = head;  s = p->next;  while(s->key <= k && s != tail) // k가 들어갈 장소를 찾음  {  p = p->next;  s = p->next;  }  r = (node\*)malloc(sizeof(node));  r->key = k;  p->next = r; // p와 s 사이에 r을 삽입  r->next = s;  return r;  } |

|  |
| --- |
| \*\*연결 리스트를 화면에 출력해주는 함수 |
| void print\_list(node\* t)  {  printf(“\n”);  while(t != tail) // 꼬리가 아닐동안 루프  {  printf(“%-8d”, t->key); // t의 키 값을 출력  t = t->next; // t는 t의 다음을 가리킴  }  } |

|  |
| --- |
| \*\*연결 리스트를 모두 삭제하는 함수 |
| node \*delete\_all(void)  {  node \*s;  node \*t;  t = head->next; // 머리 다음부터 삭제  while (t !=tail) // 꼬리 앞 까지 삭제  {  s = t; // s는 삭제할 노드를 물고 있음  t = t->next; // t는 다음 노드로  free(s); // s를 메모리에서 삭제  }  head->next = tail; // 연결을 다시 정비, 머리의 다음은 꼬리  return head;  } |

3.4.2 환영 연결 리스트(Circular Linked List)

- 단순 리스트와 비슷하지만 tail이라는 개념이 없다.

|  |
| --- |
| \*\*환영 연결 리스트 Ex요셉의 문제 |
| #include<stdio.h>  typedef struct \_node  {  int key;  struct \_node \*next;  } node;  node \*head;  void insert\_node(int k)  { // 1부터 K까지의 값을 가지는 환형 연결 리스트 구성  node \*t;  int i;  t = (node\*)malloc(sizeof(node));  t->key = 1;  head = t; // 연결 리스트의 시작점  for (I = 2; I <= k; i++)  {  t->next = (node\*)malloc(sizeof(node)); // t의 다음 노드 생성  t = t->next;  t->key = i;  }  t->next = head; // 마지막을 처음으로 물림, 환영  }  void delete\_after(node \*t)  { // t 다음의 노드를 삭제  node \*s;  s->next = t->next->next;  free(s);  }  void josephus(int n, int m)  { // 요셉의 문제를 풂, n개의 노드를 m 간격으로  node \*t;  int I;  insert\_nodes(n); // 환형 연결 리스트 구성  t = head;  printf(“\nAnswer : “);  while (t != t->next) // 연결 리스트에 노드가 남아 있을 동안  {  for (I = 0; I < m-1; i++)  t = t->next;  printf(“%d “, t->next->key);  delete\_after(t); // 출력하고 삭제  }  printf(“%d”, t->key); // 마지막 노드 출력  }  void main(void)  {  Int n, m;  printf(“\nIf you want, to quit, enter 0 or minus value”);  while (1)  {  printf(\nEnter N and M -> “);  scanf(“%d %d”, &n, &m);  if(n <= 0 || m <= 0)  return;  josephus(n, m);  }  } |

3.4.3 이중 연결 리스트(Doubly Linked List)

- 단순 연결 리스트는 가리키는 하나의 링크를 가져서 바로 전의 노드를 알 수 없는 단점이 있다.

- 이중 연결 리스트는 다음의 노드를 가리키는 링크와 전의 노드를 가리키는 링크 두 가지를 가져서 바로 전의 노드에도 접근할 수 있다.

- 하나의 링크를 더 사용하기 때문에 단순 연결 리스트보다는 노드당 2~4바이트 정도 더 소요된다.

|  |
| --- |
| \*\*이중 연결 리스트 노드 |
| Typedef struct \_dnode  {  Int key; // 정보 전달  Struct \_dnode \*prev; // 바로 전의 노드를 가리키는 링크  Struct \_dnode \*next; // 바로 뒤의 노드를 가리키는 링크  } dnode; |

- 노드를 삽입하거나 삭제할 때 많은 경우가 네 개의 링크를 조작해야 하기 때문에 조금 복잡하다.

|  |
| --- |
| \*\*이중 연결 리스트를 초기화 하는 함수 |
| dnode \*head, \*tail;  void init\_dlist(void)  {  head = (dnode\*)malloc(sizeof(dnode));  tail = (dnode\*)malloc(sizeof(dnode));  head->next = tail; // 머리의 다음은 꼬리  head->prev = head; // 머리의 앞은 머리  tail->next = tail; // 꼬리의 다음은 꼬리  tail->prev = head; // 꼬리의 앞은 머리  } |