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  } |

|  |
| --- |
| \*\*포인터 t앞에 정수 k를 가지는 노드를 삽입하는 함수 p.147 <그림 3-17> |
| dnode \*insert\_dnode\_ptr(int k, dnode \* t) // t앞에 k를 삽입  {  dnode \*I; // i는 삽입될 노드  if(t == head) // 머리 앞에는 아무것도 삽입할 수 없다.  return NULL;  i = (dnode\*)malloc(sizeof(dnode));  i->key = k;  t->prev->next = I; // t 앞 노드의 다음은 I 노드이다.  i->prev = t->prev; // I의 앞은 t의 앞 노드이다.  t->prev = I; // t의 앞은 I 노드이다.  i->next = t; // i의 다음은 t 노드이다.  return;  } |

|  |
| --- |
| \*\*포인터 p를 이중 연결 리스트에서 삭제하는 함수 \*p.148 <그림 3-18> |
| Int delete\_dnode\_ptr(dnode \*p)  {  If(p == head || p == tail) // 머리나 꼬리를 없앨 수는 없다.  Return 0;  p->prev->next = p->next; // p 앞 노드의 다음은 p 다음 노드이다.  p->next->prev = p->prev; // p 다음 노드의 앞은 p 앞 노드이다.  free(p); // p를 메모리에서 해제  return 1;  } |
| \*\*정수를 가지는 노드를 찾는 함수 |
| dnode \*find\_dnode(int k)  {  dnode \*s;  s = head->next;  while(s->key != k && s != tail) // 찾거나 꼬리이면 끝  s = s->next;  return s;  } |

|  |
| --- |
| \*\*find\_dnode()함수를 이용한 노드를 삽입, 삭제하는 함수 |
| Int delet\_dnode(int k)  {  Dnode \*s;  S = find\_dnode(k); // k의 값을 갖는 노드를 찾는다.  If (s != tail) // s가 tail이 아니면 찾은 것이다.  {  s->prev->next = s->next; // s 앞 노드의 다음은 s의 다음 노드이다.  s->next->prev = s->prev; // s 다음 노드의 앞은 s 앞 노드이다.  free(s);  return 1;  }  Return 0;  } |
| dnode \*insert\_dnode(int k, int t) // t 앞에 k를 삽입  {  dnode \*s;  dnode \*I = NULL;  s = find\_dnode(t); // t값의 노드를 찾는다.  if(s != tail) // 만약 찾았으면  {  i = (dnode\*)malloc(sizeof(dnode));  i->key = k;  s->prev->next = I; // s 앞 노드의 다음은 i노드이다.  i->prev = s->prev; // i의 앞은 s 앞 노드이다.  s->prev = I // s의 앞은 I 노드이다.  i->next = s; // i의 다음은 s 노드이다.  }  return I; // 못 찾았으면 null을 리턴한다.  } |

|  |
| --- |
| \*\*오름차순(ordered) 노드 삽입 함수 |
| dnode \*ordered\_insert(int k)  {  dnode \*s;  dnode \*I;  s = head->next;  while(s->key <= k && s != tail)  s = s->next;  i = (dnode\*)malloc(sizeof(dnode));  i->key = s;  s->prev->next = I;  i->prev = s->next;  s->prev = I;  i->next = s;  return I;  } |

- 이중 연결 리스트는 에디터, 워드프로세서 등에서 활용된다.

-> 왜냐하면, 워드프로세서의 문장들은 그 크기가 자주 변하고, 재배치되는 경우도 빈번하다.

- 이중 연결 리스트를 사용할 경우 자료의 크기 변화나 재배치에 효율적이다.

- 앞 뒤 링크를 모두 가지고 있어 문장 간의 이동도 용이하다.

3.5 단순 연결 리스트 응용 : 명함 관리

- 단순 연결 리스트는 동적인 데이터 베이스를 구축하는데 적합하다.

- 다음 프로그램은 데이터베이스는 아니지만 명함을 관리하는 간단한 프로그램을 작성

3.5.1 프로그램의 구상, 입출력 정의, 한계

- 프로그램을 작성한다는 것은 사실 끝이 없는 작업이다.

- 다음 프로그램은 단순연결리스트가 어떻게 활용되는지 보여주는 예시다.

- 자료구조를 정의하겠다.

1. 명함을 입력 받는다.(Input Namecard)

2. 명함을 삭제한다.(Delete Namecard) : 이름으로 찾음

3. 명함을 검색한다.(Search Namecard) : 이름으로 찾음

4. 디스크에서 명함들을 읽는다.(Load Namecard) : NAMECARD.DAT

5. 디스크에 명함들을 저장한다.(Save Namecard) : NAMECARD.DAT

6. 명함들의 리스트를 화면에 보여준다.(List Namecard)

7. 명함들의 리스트를 프린터로 출력한다.(Print Namecard)

8. 프로그램을 끝낸다.(End Program)

3.5.2 자료구조의 생성

- 명함관리에서 사용하는 자료 구조는 이름, 회사, 전화번호 세 가지의 필드를 갖는 단순 연결 리스트이다.

|  |
| --- |
| \*\*매크로 상수 정의, 3개의 필드를 가진 단순연결리스트 노드 정의, 초기화 함수 구현 |
| #define NAME\_SIZE 21  #define CORP\_SIZE 31  #define TEL\_SIZE 16  #define REC\_SIZE (NAME\_SIZE + CORP\_SIZE + TEL\_SIZE)  typedef struct \_card  {  char name[NAME\_SIZE]; // 이름 저장  char corp[CORP\_SIZE]; // 화사 저장  char tel[TEL\_SIZE]; // 전화번호 저장  struct \_card \*next; // 다음 노드로의 링크  } card;  card \*head, \*tail; // 머리와 꼬리  void init\_card(void)  {  head = (card\*)malloc(sizeof(card));  tail = (card\*)malloc(sizeof(card));  head->next = tail; // 머리 다음의 꼬리 노드  tail->next = tao; // 꼬리 노드 다음은 꼬리 노드  return;  } |

- REC\_SIZE 는 노드 중에서 링크(포인터)를 제외한 실질적인 자료가 담긴 크기이다.

- 디스크에 노드를 저장할 때 링크는 제외한 크기가 되어 디스크에 저장할 필요가 없는 링크는 제외할 수 있다. 🡨 이해가 안됨

3.5.3 입력, 삭제, 검색 함수

|  |
| --- |
| \*\*입력을 받는 함수 |
| void input\_card(void)  {  card \*t;  t = (card\*)malloc(sizeof(card));  printf("\nInput namecard menu : ");  printf("\n Input name -> "); // 이름을 입력받음  gets(t->corp);  printf("\n Input corporation -> "); // 회사를 입력바음  gets(t->corp);  printf("\n Input telephone number -> "); // 전화번호 입력  gets(t->tel);  t->next = head->next; // 연결리스트의 최선두에 놓음  head->next = t;  } |

- 취향에 따라 ordered\_insert()함수를 사용해도 된다.

- gets()함수는 키보드로부터 문자열을 입력받는 함수이며 종료는 ENTER키에 의하며 자동적으로 문자열에 종료 문자(NULL)를 붙여주는 편한 문자열 입력 함수다.

- 해당 프로그램은 문자열의 길이를 제안하였기 때문에 길이를 초과할 경우 심각한 논리 에러가 발생한다.

- 숙제: 문자열의 길이를 제한하여 문자열을 입력 받는 강력한 입력 함수 만들기

|  |
| --- |
| \*\*s문자열과 같은 이름을 연결 리스트에서 찾아 삭제하는 함수 |
| int delete\_card(char \*s)  {  card \*t;  card \*p;  p = head;  t = p->next;  while (strcmp(s, t->name) != 0 && t != tail) // 이름을 찾음  {  p = p->next;  t = p->next;  }  if(t == tail) // 이름을 찾지 못했다.  return 0;  p->next = t->next; // 이름을 찾은 경우 삭제, p의 다음은 t의 다음 노드  free(t);  return 1;  } |

|  |
| --- |
| \*\* 선현 검색 함수 |
| card \*search\_card(char \*s)  {  card \*t;  t = head->next;  while (strcmp(s, t->name) != 0 && t != tail)  t = t->next;  if (t == tail)  return NULL; // 찾지 못했음  else  return t;  } |

3.5.4 연결 리스트의 디스크 입출력

- 연결 리스트를 디스크에 입출력할 때 링크를 저장하지 않아야한다.

🡪 왜냐하면, 연결 리스트의 각 노드들은 메모리에 동적으로 할당되기 때문에 실행할 때마다 노드의 링크가 가리키는 주소가 달라질 수 있다.

그리고 링크는 주소의 메모리에서의 주소를 가리키기 때문에 디스크에 저장된 링크는 사실 아무 필요도 없는 정보다.

|  |
| --- |
| \*\*디스크에 명함을 저장하는 함수 |
| void save\_cards(char \*s)  {  FILE \*fp;  card \*t;  if((fp = fopen(s, "wb")) == NULL)  { // 파일 쓰기 에러, 대부분 디스크가 꽉 찼을 경우이다.  printf("\n Error : Disk write failure.");  return;  }  t = head->next; // t는 제일 처음 노드  while(t != tail)  {  fwrite(t, REC\_SIZE, 1, fp);  t = t->next; // 다음 노드로 이동  }  fclose(fp);  } |

-

|  |
| --- |
| \*\*디스크에서 읽은 노드를 메모리에 할당한 다음 연결리스트에 삽입하는 함수 |
| void load\_cards(char \*s)  {  FILE \*fp;  card \*t;  card \*u;  if ((fp == fopen(s, "rb")) == NULL)  {  printf("\n Error : %s is not exist.", s);  return;  }  t = head->next; // 디스크에서 읽기 전에 모든 노드를 다 지운다.  while(t != tail)  {  u = t;  t = t->next;  free(u);  }  head->next = tail;  while (1)  {  t = (card\*)malloc(sizeof(card));  if (!fread(t, REC\_SIZE, 1, fp)) // 파일의 끝이면  {  free(t);  break;  }  t->next = head->next;  head->next = t;  }  fclose(fp);  } |

- fread()함수는 레코드의 수를 리턴한다. 레코드의 수가 0이면 파일의 끝을 나타낸다.

- malloc()함수 보다 calloc()함수를 사용이 더 클린하다.

3.5.5 방향 재설정(Redirection)을 이용한 출력 함수

- 방향 재설정이란 출력의 방향을 임의로 바꿀 수 있음을 의미한다.

3.6 이중 연결 리스트 응용: 텍스트뷰어

`- 텍스트뷰어란 텍스트 파일을 간편한 방법으로 보기 위한 프로그램이다.

- 대표적으로 BORLAND사의 readme.com 프로그램이 있다.

3.6.1 설계와 한계

1. 이중 연결 리스트를 초기화하는 모듈

2. 파일에서 문장들을 읽어 이중 연결 리스트를 구축하는 모듈

3. 이중 연결 리스트의 내용을 화면에 출력하는 모듈

4. 사용자의 키 입력을 처리하는 모듈로 구성

3.6.2 자료 구조의 정의

|  |
| --- |
| \*\*이중연결리스트 노드 |
| typedef struct \_line  {  char \*buf;  struct \_line \*prev;  struct \_line \*next;  } line; |

|  |
| --- |
| \*\*전역변수들의 정의 |
| line \*head, \*tail; // 머리와 꼬리 노드  int total, now; // 파일의 총 라인 수와 현재의 라인 번호  char filename[13]; // 파일의 이름을 저장 |

|  |
| --- |
| \*\*초기화 함수 |
| void init\_line(void)  {  head = (line\*)malloc(sizeof(line));  tail = (line\*)malloc(sizeof(line));  head->next = tail;  head->prev = head;  tail->next = tail;  tail->prev = head;  } |

3.6.3 문자열을 파일에서 읽고 저장하기

1. 파일을 연다(open).

2. 전체 라인 수를 0으로 초기화한다.

3. 파일의 끝에 도달할 때까지

3.1 파일에서 한 라인을 읽어서 buf에 저장한다.

3.2 buf에 저장된 문장의 길이가 80이 넘으면 자른다.

3.3 새로운 노드를 생성하고 여기에 buf에 저장된 문자열을 복사한다.

3.4 새로 생성된 노드를 이중 연결 리스트의 제일 뒤에 삽입한다.

3.5 전체 라인 수를 증가시킨다.

3.6 3으로 돌아간다.

4. 파일을 닫고 끝낸다.

|  |
| --- |
| \*\*파일을 읽어오는 함수 |
| void load\_file(void)  {  FILE \*fp;  char buf[256]; // fgets()를 위해서 여유를 많이 둔다.  line \*t;  if((fp = fopen(filename, "rt")) == NULL) // 파일 이름이 틀리면  {  printf("\n Error : Can't read that file.");  exit(1);  }  total = 0;  printf("\n File loading...");  while (!feof(fp)) // 파일 이름이 틀리면  {  fgets(buf, 255, fp); // 파일에서 한 문장 읽음  if(strlen(buf) > 80) // 한 라인을 80자로 자름  buf[80] = 0;  if((t = (line\*)malloc(sizeof(line))) == NULL) // 노드 생성  {  printf("\n Error : Out of Memory");  exit(1);  }  if((t->buf = (char\*)malloc(strlen(buf))) == NULL)  { // 문자열을 저장할 공간 확보  printf("\n Error : Out of Memory");  exit(1);  }  strcpy(t->buf, buf); // 입력된 문자열을 복사  // 생성된 노드를 제일 뒤에 삽입  t->prev = tail->prev; // t의 앞은 꼬리의 앞 노드  t->next = tail; // t의 다음은 꼬리 노드  tail->prev->next = t; // 꼬리 앞 노드의 다음은 t 노드  tail->prev = t; // t의 앞노드는 t노드  total++; // 전체 라인 수 증가  }  fclose(fp); // 파일 닫음  } |

3.6.4 화면 출력 함수

- 텍스트뷰어의 화면은 상태 라인(Status line)과 실제 파일 내용들이 출력된다.

|  |
| --- |
| \*\*상태 라인을 출력하는 함수 |
| void show\_header(void)  {  textattr((LIGHTGRAY<<4) | BLACK); // 반전  gotoxy(1, 1); // 커서를 화면 좌측 상단으로  cprintf(" TVIEW : %-12s Loc : %6d of %6d By Lee jaekyu ",  filename, now, total);  textattr((BLACK<<4) | LIGHTGRAY); // 보통으로  } |

- textattr() 함수는 문자의 속성을 반전한다.

|  |
| --- |
| \*\*실제로 파일의 내용을 상태 라인 밑에다 페이지 출력하는 함수 |
| void( show\_page(line \*t))  {  int i=0;  clrscr(); // 화면을 지운다.  show\_header(); // 상태 라인을 출력  gotoxy(1, 2); // 상태 라인 바로 아래에 커서를 옮김  while(i++ < 23 && t != tail)  { // 23라인을 다 출력하거나 마지막 라인을 넘어서면 중지  cprintf("%-80s\r", t->buf);  t = t->next; // t는 다음 노드로 이동  }  } |

의문 1) 텍스트 화면은 모두 25라인을 출력할 수 있다.

한 라인은 상태 라인으로 사용하므로 24라인을 출력해야 하는데 23라인을 출력한다.

의문 2) cprintf문에서 개행을 위해서 \n\r을 사용해야 하는데 왜 \r만 사용하는가?

\*p.176 참조 BIOS의 이야기가 진득하게 나온다.

3.6.5 키 조작 함수

- PGDN, PGUP, ESC 이벤트 구현

- 포인터의 내용을 참조에 의한 호출로서 함수 내에서 변경하고자 할 때에는 포인터의 포인터를 사용해야 한다.

|  |
| --- |
| \*\*처음 라인 또는 끝 라인까지 만 이동하는 함수 |
| void move\_line(int d, line \*\*t)  {  if(d < 0)  while (d++ != 0 && (\*t)->prev != head)  {  \*t = (\*t)->prev;  now--;  }  else  while (d-- != 0 && (\*t)->next != tail)  {  \*t = (\*t)->next;  now++;  }  } |

|  |
| --- |
| \*\*사용자 키 입력 처리 |
| void key\_proc(void)  {  line \*t;  int key; // 키 입력을 받을 변수  now = 1; // 현재의 라인은 1  t = head->next; // t는 제일 처음 노드  show\_page(t); // 화면을 출력, 제일 첫 페이지  while ((key = bioskey(0)) != ESC) //ESC이면 끝  {  switch (key)  {  case PGUP : move\_line(-23, &t); //23라인 앞으로 이동  show\_page(t);  break;  case PGDN : move\_line(+23, &t); //23라인 뒤로 이동  show\_page(t);  break;  }  }  clrscr(); // 화면 지우기  } |

3.6.7 계선할 점 p.183 -미래의 나의 숙제

- 출력 함수 계선

- 한 줄씩 이동하는 기능 추가

- 좌우 스크롤 추가

- 그 외 다양한 기능

- 한글 출력 라이브러리

3.7 스택(Stack)

- 스택(Stack)이나, 큐(Queue), 트리(Tree)와 같은 자료 구조는 자신이 행위적 측면을 포함하는 자료 구조다.

- 제한된 접근 방식을 규정하고 있으며 이를 응용한 알고리즘도 매우 다양하다.

Ex) …

- 스택(Stack)은 아주 중요한 자료 구조이며 시스템 내부의 기본 동작에서부터 고급 알고리즘까지 다양하게 활용되고 있다.

3.7.1 스택의 개념

- 입 \* 출구가 같다.

--> 먼저 들어간 것은 밑에 있게 되고 나중에 들어간 것이 위에 있는 형태

🡪 따라서, 제일 나중에 들어간 것이 제일 먼저 나오게 된다.

- LIFO(Last In First Out) 구조이다.

스택의 동작

- 푸시(Push): 스택의 상단에 값을 집어 넣는 동작.

- 팝(Pop): 스택에서 값을 하나 빼내는 동작

3.7.2 배열로 구현하는 스택

|  |
| --- |
| \*\* 배열로 구현한 스택의 자료구조 |
| #define MAX 10  int stack[MAX]; // 스택의 긴통  int top; // 스텍의 상단 |

|  |
| --- |
| \*\*초기화 함수 |
| void init\_stack(void)  {  top = -1;  } |

- 다음은 푸시(Push) 동작의 정의이다.

- top를 하나 증가한다.

- stack[top]에 값을 저장한다.

- 스택 오버플로(Stack Overflow)가 일어날 경우 에러를 처리한다.

|  |
| --- |
| \*\* 푸시(push) 함수 |
| int push(int t)  {  if(top >= MAX - 1) // 스택이 꽉찼는가?  {  printf("\n Stack overflow.");  return -1; // 에러 처리  }  stack[++top] = t; // top을 증가시키고 t를 저장  return t;  } |

- 반대로 팝(Pop)에 대한 동작이다.

- top를 하나 감소시킨다.

- stack에 값을 return한다.

- 스택 언더플로(Stack Underflow)가 일어날 경우 에러를 표시한다.

|  |
| --- |
| \*\*팝(Pop) 함수 |
| int pop(void)  {  if(top < 0) // 스택이 텅 비었는가?  {  printf("\n Stack underflow.");  return -1; // 에러 표시  }  return stack[top--]; // 스택 상단의 값 리턴하고 top 감소  } |

- 스택(Stack)은 배열과 top이라는 상단의 위치, 그리고 push()함수와 pop()함수만 있으면 스택을 완전히 구성이 된다.

3.7.3 연결 리스트를 이용한 스택의 구현

- 연결 리스트 자체가 동적인 할당을 통해서 구현되기 때문에 연결 리스트를 이용해서 구현되는 스택은 매우 유연한다.

**연결 리스트를 이용한 스택 구현의 장점**

**-** 자료만큼만 메모리를 잡아먹기 때문에 메모리가 절약된다.

- 스택의 크기가 메모리가 허용하는 한도에서 커질 수 있다.

- 스택은 입\*출구가 하나이기 때문에 단순 연결 리스트로 쉽게 구현 가능하다.

|  |
| --- |
| \*\*단순 연결 리스트로 구성한 스택 자료구조 \*p.191 <그림 3-20> |
| typedef struct \_node  {  int key;  struct \_node \*next;  } node;  node \*head, \*tail; |

|  |
| --- |
| \*\* 초기화 함수; |
| void init\_stack(void)  {  head = (node\*)malloc(sizeof(node));  tail = (node\*)malloc(sizeof(node));  head->next = tail;  tail->next = tail;  } |

- 다음은 스택의 푸시 동작이다.

- 연결 리스트를 이용한 스택 구현에서는 푸시 동작은 스택 오버플로(Stack Overflow)라는 개념이 없다.

-> 미리 정해진 스택의 크기가 없고 스택이 점점 커지는 형태이기 때문

-> 하지만 메모리가 부족해서 더 이상 새로운 노드를 만들 수 없을 때 메모리가 부족하는 에러를 내도록 만든다.

- head->next가 스택의 상단(Top)이 되도록 구현한다.

|  |
| --- |
| \*\* 단순 연결 리스트를 이용한 푸시(Push)동작 |
| int push(int k)  {  node \*t;  if((t = (node\*)malloc(sizeof(node))) == NULL)  { // 메모리가 부족할 경우 에러를 낸다.  printf("\n Out of memory... ");  return -1;  }  t->key = k;  t->next = head->next; // t를 머리의 다음에 삽입한다.  head->next = t;  return k;  } |

- 팝 동작은 상단이 head->next임으로 head->next를 삭제한다.

- 스택이 텅 비었을 경우 스택 언더플로(Stack underflow) 에러 표시

- head->next == tail 일 경우 언더플로

|  |
| --- |
| \*\*단순 연결 리스트를 이용한 팝(pop)동작 |
| int pop(void)  {  node \*t;  int i;  if(head->next == tail)  {  printf("\n Stack underflow.");  return -1;  }  t = head->next; // head->next 노드를 삭제하기 위해 t에 물림  i = t->key;  head->next = t->next;  // head->next 노드를 연결 리스트에서 제외시킴  free(t); // t를 삭재  return i; // 삭제된 t의 값을 리턴  } |

- 스택의 내용을 모두 비우는 clear\_statck()함수 구현

|  |
| --- |
| \*\* |
| void clear\_stack(void)  {  node \*t, \*s;  t = head->next; // 첫 노드를 t에 물린다.  while (t != tail) // 끝 노드까지 루프  {  s = t; // 노드를 삭제하기 위해 s에 물림  t = t->next; // t는 다음 노드로 건너감  free(s); // 물린 노드 해제  }  head->next = tail;  } |

3.7.4 정리

**배열(Array)**

- 친숙한 데이터형이기 때문에 프로그래밍에 부담이 없다.

- 스택의 내용이 얼마 없을 때 엄청난 메모리 낭비가 된다.

**연결 리스트(Linked List)**

- 링크에 관한 정보를 따로 저장함이 없기 때문에 속도도 빠르고 부가적인 정보를 필요로 하지 않는 장점이 있음

- 포인터를 사용하기 대문에 프로그래밍에 부담이 있음

- 스택의 자료 수 만큼의 노드만 메모리를 차지하고 스택의 크기가 정해지지 않아 사실상 메모리의 한계까지 스택의 크기를 늘릴 수도 있고 스택의 크기를 아주 작게 줄일 수도 있다.

🡪 이것이 동적인 자료구조의 이점이다.

미래의 나의 숙제

- 배열로 동적 할당 구현 p.197

3.7.5 스택의 활용

- 스택은 시스템 내부에서부터 고급 알고리즘까지 매우 다양한 방법으로 사용된다.

- 스택은 주 용도는 자료의 임시 저장 장소로 쓰인다.

- 시스템 내부에 위치한 내부 스택(Internal Stack)

-> 함수의 호출이나 인터럽트의 처리시에 현재의 주소나 상태를 임시로 저장해두는데 사용한다.

- 고급 알고리즘

🡪 산술식 계산, 재귀 호출을 비 재귀 호출로 바꾸는 경우 등등 사용

3.8 스택의 응용: CALC 유틸리티 작성

- CALC 유틸리티를 작성한다.

- CALC는 중위표기법(Infix notation)의 수식을 인자로 받아서 읽어서 후위표기법(Postfix notation)으로 바꾸어서 이 후위 표기법의 수식을 스택을 이용하여 값을 연산하여서 화면에 출력한다.

단, CALC는 연산자의 우선순위에 의거하여 괄호를 사용한 중위표기 수식을 인자로 받는다.

또한, 간단한 정수와 사칙연산 밖에 하지 못한다.

3.8.1 수식의 표기법  
- 중위표기법(Infix noation): 수식의 표현 방법

* 중위표기법이라고 명명한 이유
* 연산자(operator)가 두 피연산자(Operand)의 사이(in)에 들어가기 때문이다.

Ex) A \* B

- 전위표기법(Prefix notation)

Ex) \* A B

- 후위표기법(Postfix notation)

Ex) A B \*

- 중위표기법을 후위표기법으로 바꾸는 이유

🡪 중위 표기법은 괄호를 반드시 사용해야한다.

Ex) A\*B + C/D – E 🡪 (A\*B) + (C/D) – E

3.8.2 중위표기법을 후위표기법으로 변환하는 방법 1

- 중위표기법을 후위표기법으로 변환할 때 스택을 사용한다.

- 후위표기법 알고리즘 변환

1. “(“ 문자는 무시하고 넘어간다.

2. 피 연산자는 그대로 출력한다.

3. 연산자는 스택에 푸시(Push)한다.

4. “)”를 만나면 스택에서 팝(Pop)하여 출력한다.

\*p. 200 그림 참고

- dst: 후위표기법으로 변환한 식을 저장한다.

- src: 중위표기법의 수식이 저장되어 있음

|  |
| --- |
| \*\* 후위표기법으로 변환 알고리즘1 |
| void postfix1(char \*dst, char \*src)  {  char c;  init\_stack(); // 스택을 초기화  while (\*src) // 중위표기법의 수식이 남아있는 동안  {  if(\*src == ')') // ')'를 만나면 푸시되어 있던 연산자를 팝하여  { // dst의 현 위치에 저장한다.  \*dst++ = pop();  \*dst++ = ' '; // 문자의 구분을 위해 공백을 출력  src++;  }  else if (\*src == '+' || \*src == '-' || \*src == '\*' || \*src == '/')  { // 연산자이면 연산자를 스택에 푸시한다.  push(\*src);  src++;  }  else if (\*src >= '0' && \*src <= '9')  { // 숫자는 피 연산자이다. 숫자의 연속을 읽어들여 dst에  do // 그대로 복사  {  \*dst++ = \*src++;  } while (\*src >= '0' && \*src <= '9');  \*dst++ = ' ';  }  else // 이것도 저것도 아니면 무시하고 src를 하나 증가  src++;  }  \*dst = 0; // 후기표기법의 문자열의 뒤에 NULL 문자를 붙여준다.  } |

3.8.3 중위표기법을 후위표기법으로 변환하는 방법2

- 중위표기법에 완전히 괄호를 씌워야 한다는 단점이 있다.

🡪 즉, 연산자의 우선순위를 전혀 고려하지 않았다는 뜻이다.

* 따라서, 연산자의 우선순위를 고려하여 변환하는 방법을 생각해보자

-다음은 우선순위를 고려한 중위표기법에서 후위표기법으로 전환하는 알고리즘이다.

1. ‘(‘를 만나면 스택에 푸시한다.

2. ‘)’를 만나면 스택에서 ‘(‘가 나올 때까지 팝하여 출력하고 ‘(‘는 팝하여 버린다.

3. 연산자를 만나면 스택에서 그 연산자보다 낮은 우선순위의 연산자를 만날 때까지 팝하여 출력한 뒤에 자신을 푸시한다.

4. 피 연산자는 그냥 출력한다.

5. 모든 입력이 끝나면 스택에 있는 연산자들을 모두 팝하여 출력한다.

p.203 <그림 3-21> 참고

- 다음은 스택의 최상단값을 리턴하는 함수이다.

|  |
| --- |
| \*\* |
| int get\_stack\_top(void)  {  return (tcp < 0) ? -1 : stack[top];  } |

- 주어진 문자가 연산자인지 판별하는 함수

|  |
| --- |
| \*\* |
| int is\_operator(int k)  {  return (k == '+' || k == '-' || k == '\*' || k == '/');  } |

- 연산자의 우선순위를 수치로 변환해주는 함수

|  |
| --- |
| \*\* |
| int precedenc(int op)  {  if(op == '(') return 0;  if(op == '+' || op == '-') return 1;  if(op == '\*' || op == '/') return 2;  else return 3;  } |

- 중위표기법을 후위표기법으로 바꾸어주는 함수

|  |
| --- |
| \*\* |
| void postfix(char \*dst, char \*src)  {  char c;  init\_stack(); // 스택의 초기화  while(\*src)  {  if(\*src == '(')  {  push(\*src);  src++;  }  else if(\*src == ')')  {  while(get\_stack\_top() != '(')  {  \*dst++ = pop();  \*dst++ = ' ';  }  pop();  src++;  }  else if(is\_operator(\*src))  {  while (!is\_stack\_empty() &&  precedenc(get\_stack\_top()) >= precedence(\*src))  { // 우선순위가 높은 연산자들을 모두 팝  \*dst++ = pop();  \*dst++ = ' ';  }  push(\*src); // 그리고 나서 푸시  src++;  }  else if(\*src >= '0' && \*src <= '9');  { // 피연산자는 그냥 출력  do  {  \*dst++ = \*src++;  } while(\*src >= '0' && \*src <= '9');  \*dst++ = ' ';  }  else  src++;  }  while (!is\_stack\_empty())  { // 모두 끝났으면 스택에 있는 모든 내용을 팝  \*dst++ = pop();  \*dst++ = ' ';  }  dst--;  \*dst = 0;  } |

3.8.4 후위표기법 수식의 평가

- 후위표기법 수식의 평가도 역시 스택을 이용한다.

- 후위표기법 수식을 계산하는 알고리즘

1. 숫자를 만나면 숫자는 스택에 푸시한다.

2. 연산자를 만나면 스택에서 팝을 두 번하여 그 두 데이터를 가지고 연산한 다음 그 결과를 스택에 다시 푸시한다.

p.206 글과 그림 참고

|  |
| --- |
| \*\*중위표기법의 수식의 문자열을 읽어서 후위표기법으로 변경한 다음 수식을 평가하여 결과를 리턴 |
| void postfix(char \*dst, char \*src)  {  char c;  init\_stack(); // 스택의 초기화  while(\*src)  {  if(\*src == '(')  {  push(\*src);  src++;  }  else if(\*src == ')')  {  while(get\_stack\_top() != '(')  {  \*dst++ = pop();  \*dst++ = ' ';  }  pop();  src++;  }  else if(is\_operator(\*src))  {  while (!is\_stack\_empty() &&  precedenc(get\_stack\_top()) >= precedence(\*src))  { // 우선순위가 높은 연산자들을 모두 팝  \*dst++ = pop();  \*dst++ = ' ';  }  push(\*src); // 그리고 나서 푸시  src++;  }  else if(\*src >= '0' && \*src <= '9');  { // 피연산자는 그냥 출력  do  {  \*dst++ = \*src++;  } while(\*src >= '0' && \*src <= '9');  \*dst++ = ' ';  }  else  src++;  }  while (!is\_stack\_empty())  { // 모두 끝났으면 스택에 있는 모든 내용을 팝  \*dst++ = pop();  \*dst++ = ' ';  }  dst--;  \*dst = 0;  }  int calc(char \*p)  {  int i;  init\_stack(); // 스택을 초기화  while(\*p)  {  if (\*p >= '0' && \*p <= '9')  {  i = 0;  do  {  i = i \* 10 + \*p - '0';  p++;  } while (\*p >= '0' && \*p <= '9');  push(i); // 피연산자는 스택에 푸시  }  else if(\*p == '+')  { // 연산자를 만나면 스택에서 팝하여 연산하고 푸시  push(pop() + pop());  p++;  }  else if(\*p == '\*')  {  push(pop() \* pop());  p++;  }  else if(\*p == '-')  { // 교환법칙이 성립하지 않는 연산자들  i = pop();  push(pop() - i);  p++;  }  else if(\*p == '/')  {  i = pop();  push(pop()/i);  p++;  }  else  p++;  }  return pop();  } |

- 숫자열을 읽어서 정수로 변환하는 방법

- +\*연산자와는 다른 방식으로 -/연산자를 처리하는 것

3.8.5 CALC의 완전한 리스트

실행 방법 CALC.exe 2\*3+6/2-4

3.9 큐(Queue)

- 큐는 먼저 들어온 것이 먼저 나간다.

- FIFO(First In First Out) 구조이다.

3.9.1 큐의 개념