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

- 큐는 입구와 출구가 따로 있는 긴 통이라고 생각하면 쉽다.

- 큐는 접근이 제한된 자료구조이며 행위적 측면을 부여받은 추상적 자료형이기 때문에 큐를 조작하는 방법은 두가지로 제한된다.

- 큐에 자료를 집어넣을 때는 뒤(rear)에서 집어넣는다.

-> 이 집어넣는 동작을 put 동작이라고 한다.

- 큐에서 자료를 얻어낼 때는 앞(front)에서 얻어낸다.

-> 이 자료를 얻는 동작을 get 동작이라고 한다.

3.9.2 배열을 이용한 큐의 구현

- 다음은 큐의 구조 정이다.

|  |
| --- |
| #define MAX 10  int queue[MAX];  int front, rear; |

- 다음은 큐의 정의를 이용한 put 기능과 get 기능을 하는 함수이다.

|  |
| --- |
| int put(int k)  {  // rear가 한계를 넘지 않았는가? 넘었으면 오버플로 에러 처리  queue[++rear] = k;  }  int get(void)  {  // 큐가 비어있지 않은가? 비어있다면 언더플로 에러 처리  return queue[front++];  } |

- 문제점

1. 큐에 자료를 집어넣고 빼는 동작을 계속하다보면 rear와 front는 계속 증가됨

🡪 오버플로 에러 발생 🡪 따라서 배열에 끝에 이르렀는지 점검해서 배열에 끝에 닿았을 경우 큐의 내용 전체를 복사해서 배열의 앞부분으로 옮기는 동작이 필요하다.

\*p.216그림 참고

🡪 다음과 같은 문제점을 해결하는 방법이 원형 큐(circular Queue)이다.

**원형 큐란?**

- 배열의 첫 요소와 마지막 요소가 붙어서 마치 뱀의 머리가 꼬리를 물고 있는 듯한 모양

- 원형의 자료구조는 끝이라는 개념이 없어서 계속 돌기만 한다.

\*p.217참고

- 주의할점은 rear가 큐의 끝 다음의 빈공간을 가리키게 한다.

🡪 그러면 front = rear일 경우 큐가 비어있는지 아니면 가득차있는지 알 수 없다.

- 이을 구별하기 위해 원형 큐는 완충 작용을 하는 하나의 공간을 배치한다.\*p.218 참고

|  |
| --- |
| \*\* 원형 큐(circle queue) 초기화함수 |
| void init\_queue(void)  {  front = rear = 0  } |

|  |
| --- |
| \*\*큐를 비워주는 함수 |
| void clear\_queue(void)  {  front = rear;  } |

|  |
| --- |
| \*\*put동작 함수 |
| int put(int k)  {  if((rear + 1) % MAX == front) // 큐가 꽉 찼는지 확인  {  printf("\n Queue overflow");  return -1;  }  queue[rear] = k; // rear는 빈 공간을 가리키므로 바로 k를 저장한다.  rear = ++rear % MAX; // rear를 다음으로  return k;  } |

- % 연산을 눈여겨보자!

|  |
| --- |
| \*\*get동작 함수 |
| int get(void)  {  int i;  if(front == rear) // 큐가 비어있는가?  {  printf("\n Queue underflow");  return -1;  }  i = queue[front]; // front의 값을 취득  front = ++front % MAX; // front를 다음 요소로  return i;  } |

- 큐가 비어있는지를 우선 확인한다.

🡪 front와 rear가 같은지 확인한다.

- 값을 얻기 위해 queue[front]가 취득할 값을 가져오고 front를 1증가후 배열의 크기로 %연산을 한다.

3.9.3 연결 리스트를 이용한 큐의 구현

- 연겨리스트로 큐를 구현하면 메모리의 한계까지 큐의 크기를 늘릴 수 있고, 메모리가 아주 작을 때에도 조금 밖에 차지하지 않는다.

- 큐는 이중연결리스트를 사용하는 것이 자연스럽다.

- 이중 연결 리스트를 이용해 큐를 구현하는 노드를 정의해보자

|  |
| --- |
| \*\* 이중 연결 리스트 노드 정의 |
| typedef struct \_dnode  {  int k;  struct \_dnode \*prev;  struct \_dnode \*next;  } dnode;  dnode \*head, \*tail; |

|  |
| --- |
| \*\* 큐(Queue) 초기화 함수 |
| void init\_queue(void)  {  head = (dnode\*)malloc(sizeof(dnode));  tail = (dnode\*)malloc(sizeof(dnode));  head->prev = head;  head->next = tail;  tail->prev = head;  tail->next = tail;  } |

|  |
| --- |
| \*\*put 함수 |
| int put(int k)  {  dnode \*t;  if((t = (dnode\*)malloc(sizeof(dnode))) == NULL)  { // 메모리가 다 되었으면  printf("\n Out of memory");  return -1;  }  t->key = k;  tail->prev->next = t; // t를 꼬리의 앞에 삽입한다.  t->prev = tail->prev;  tail->prev = t;  t->next = tail;  return k;  } |

|  |
| --- |
| \*\*get 함수 |
| int get(void)  {  dnode \*t;  int i;  t = head->next; // 머리의 다음 노드  if(t == tail) // 큐가 비어 있으면 언더플로  {  printf("\n Queue underflow.");  return -1;  }  i = t->key; // 리턴값을 물려둠  head->next = t->next; // 머리 다음 노드를 삭제  t->next->prev = head;  free(t); // 메모리에서 해제  return i; // 물려둔 값을 리턴  } |

|  |
| --- |
| \*\*모든 내용 삭제 함수 |
| void clear\_queue(void)  {  dnode \*t;  dnode \*s;  t = head->next;  while(t != tail) // 꼬리가 나올 때까지  {  s = t; // 삭제를 위해서 물려둠  t = t->next; // t는 다음으로 넘어감  free(s); // 삭제  }  head->next = tail; // 머리와 꼬리를 서로 맞물림  tail->prev = head;  } |

3.9.4 정리

- 큐를 구현하는 방법으로 배열을 이용하여 원형 큐를 구현하는 방법과 이중 연결 리스트를 이용하여 큐를 구현하는 방법을 알아봤다.

- 배열

장점: 배열을 이용하면 프로그래밍의 부담이 없다.

단점: 배열의 크기를 미리 정해놓아야 한다.

- 연결리스트

장점: 동적 할당으로 인해서 필요한 만큼의 메모리만을 소요하며 그 크기가 메모리의 한계까지 커질 수 있다

단점: 프로그래밍의 부담과 이중 연결 리스트로 링크 정보를 두개나 더 사용해야 한다.

3.9.5 큐의 응용

- 큐는 스택과 마찬가지로 자료의 임시 저장용으로 많이 쓰인다.

Ex) 은행의 창구,

- 시스템 내부에서도 사용된다.

Ex) 키보드 버퍼(BIOS) -> 원형 큐

그 외로 마우스 이벤트 큐(event queue)

마우스 이벤트 핸들러와 결합하여 인터럽트 방식의 마우스 제어를 가능케한다.

3.10 나무(Tree)

- 나무는 비선형 구조(Non-linear structure)이다.

Ex) MS-DOS의 디렉토리 구조, 봉황기 고교야구 대회의 대전표, 회사의 기구도, 한 가문의 족보를 그린 것 등등,

- 나무 구조는 나무의 모양을 마치 거꾸로 뒤집어 놓은 모양이다.

- 뿌리(root), 가지(link), 잎(leaf)으로 구성

3.10.1 나무 구조에서 사용하는 용어들

- 나무 구조는 어떠한 조건을 만족하는 노드(node)와 링크(link)의 집합이다.

- 노드 -> 버텍스(Vertex), 링크 -> 에지(edge)

- 노드는 어떤 정보(information)를 담고 있으며, 링크는 노드의 연결을 나타낸다.

- 경로(path)는 나무 내에서 링크에 의해 연결된 일련의 노드의 집합이다.

-> 나무를 링크에 의해 연결된 노드를 통해 연결된 노드를 통해 이동할 때 이동 경로

p.231 글 읽기

3.10.2 나무의 성질

1. 나무 구조에서 한 노드에서 다른 노드로 가는 경로(path)는 유일하다.

- 나무를 타는 경로가 중복됨이 없고, 되돌아감이 없다면 두 노드간의 경로는 반드시 한 노드에서 최소 공통 선조까지 올라갔다가 다시 다른 노드로 내려오는 유일한 경로 밖에 존재하지 않는다.

2. N개의 노드를 같은 나무는 N-1개의 링크를 가진다.

3. 꽉 찬 이진 나무(ful binary tree)의 경우 N개의 노드를 가진 경우 높이는 [log2N]+1이다. 234쪽 수식 참고

3.10.3 이진 나무 구조의 표현

- 이진 나무 구조의 표현은 배열을 이용하는 방법과 연결 리스트를 이용하는 방법이 있다.

- 배열은 프로그래밍이 쉽지만 이진 나무가 균형이 잡혀있지 않은(Unbalanced) 경우는 메모리 낭비가 심하다.

- 연결 리스트를 이용하는 방법은 이중 연결 리스트와 마찬가지로 오른쪽과 왼쪽 자식 노드에 대한 링크를 유지함으로써 구현된다.

- 연결 리스트는 균형잡힌(balanced)나무의 경우에는 배열을 이용하는 방법보다 효율이 떨어진다.

|  |
| --- |
| \*\* 나무 노드 정의 |
| typedef struct \_node  {  int key;  struct \_node \*left;  struct \_node \*right;  } node;  node \*head, \*tail; |

|  |
| --- |
| \*\* 초기화 함수 |
| void init\_trr(void)  {  head = (node\*)malloc(sizeof(node));  tail = (node\*)malloc(sizeof(node));  head->left = tail; // 모두 꼬리 노드를 향하게 한다.  head->right = tail;  tail->left = tail;  tail->right = tail;  } |

3.10.4 이진 나무타기: 순회(Tree traverse)

- 모든 노드들을 한 번 씩 중복없이 순회하는 방법

- 뿌리를 타는 첫번째 방법

1. 뿌리를 먼저 방문한다.

2. 왼쪽 작은 나무(subtree)를 방문한다.

3. 오른쪽 작은 나무를 방문한다.

- 전위순회(Treorder traverse)

|  |
| --- |
| void preorder\_traverse(node \*t)  {  if(t != tail)  {  visit(t); // 뿌리 방문  preorder\_traverse(t->left); // 왼쪽 작은 나무를 방문  preorder\_traverse(t->right); // 오른쪽 작은 나무 방문  }  } |

p.238 그림참고

- 중위순회(Inorder traverse)

1. 왼쪽 작은 나무를 방문한다.

2. 뿌리를 방문한다.

3. 오른쪽 작은 나무(subtree)를 방문한다.

|  |
| --- |
| void inorder\_traverse(node \*t)  {  if(t != tail)  {  inorder\_traverse(t->left); // 왼쪽  visit(t); // 뿌리  inorder\_traverse(t->right); // 오른쪽  }  } |

p.241

- 후위순회(Postorder traverse)

1. 왼쪽 작은 나무를 방문한다.

2. 오른쪽 작은 나무를 방문한다.

3. 뿌리를 방문한다.

|  |
| --- |
| void postorder\_traverse(node \*t)  {  if(t != tail)  {  postorder\_traverse(t->left);  postorder\_traverse(t->right);  visit(t);  }  } |

- 층별순회(Lever order traverse)

1. 큐에 뿌리노드를 put 한다.

2. 큐가 비어있지 않으면

- 1. 큐에서 get하여 t에 대입, t를 방문한다.

- 2. T의 왼쪽 자식이 있으면 그것을 큐에 put한다.

- 3. T의 오른쪽 자식이 있으면 그것을 큐에 put한다.

3. 끝

|  |
| --- |
| void levelorder\_traverse(node \*t)  {  put(t);  while(!is\_queue\_empty())  {  t = get();  visit(t);  if(t->left != tail)  put(t->left);  if(t->right != tail)  put(t->right)  }  } |

3.10.5 정리

나무는 이후 단원에서 학습하면서 완벽이해스 하자

3.11 나무 구조의 응용: 수식 나무(Parse tree)

- 수식 나무는 연산자를 뿌리로 하고 왼쪽 피연산자를 왼쪽 자식 나무에 오른쪽 피연산자를 오른쪽 자식 나무에 놓은 구조이다.

3.11.1 후위표기법에서 수식 나무를 구성하는 방법

1. 피연산자를 만나면 노드를 생성하여 스택에 푸시한다.

2. 연산자를 만나면 노드를 생성하여

1. 스택에서 팝한 노드를 오른쪽 자식으로 할당한다.

2. 스택에서 또 팝한 노드를 왼쪽 자식으로 할당한다.

3. 그리고 연산자 노드를 스택에 푸시한다.

3. 스택에서 마지막으로 남은 노드가 뿌리 노드이다.

|  |
| --- |
| node \*make\_parse\_tree(char \*p)  { // 후위표기법 수식 p를 읽어 수식 나무를 구성  node \*t;  while(\*p)  {  while(\*p == ' ') // 공백 문자는 제거  p++;  t = (node\*)malloc(sizeof(node));  t->key = \*p;  t->left = tail;  t->right = tail;  if(is\_operator(\*p)) // 연산자이면 팝을 하여 자식 노드로 삼음  {  t->right = pop();  t->left = pop();  }  push(t);  p++;  }  return pop();  } |

- 수식이 적법한지 아닌지를 판별하는 함수

|  |
| --- |
| int is\_legal(char \*s)  {  int f = 0;  while (\*s)  {  while (\*s == ' ') // 공백제거  s++;  if (is\_operator(\*s))  f--; // 연산자이면 감소  else  f++; // 피연산자이면 증가  if (f < 1) break; // f가 1보다 작아지면 언더플로  s++;  }  return (f == 1); // 피연산자수 - 연산자수 = 1이 되어여함  } |

3.11.2 완전한 리스트

-- 끝 –

4 재귀 호출(Recursion)

4.1 자기자신을 호출한다.

- 재귀 함수(Recursive Function): 자기 자신으로부터 호출이 되는 함수

- 재귀 호출(Recursion): 함수 호출

- 재귀적 언어(Recursive Language): 재귀 호출을 지원하는 컴퓨터 언어

- 순환적 언어(Iterative Language): 재귀 호출을 지원하지 않는 컴퓨터 언어

4.1.1 잘못된 재귀 호출

|  |
| --- |
| void rose(void)  {  rose();  } |

|  |
| --- |
| void cheol(void)  { // 간접 재귀함수  young();  }  void young(void)  {  cheol();  } |

- 시스템 오버플로로 시스템이 다운된다.

4.1.2 재귀 호출의 요건

- 재귀 호출이 이루어질 때마다 문제는 점점 작아져야 하며, 또한 재귀 호출이 끝이 나는 종료 조건(Terminate conditon)이 있어야 한다.

4.1.3 누승을 구하는 재귀 함수

- 누승(factorial)을 계산하는 프로그램

|  |
| --- |
| int factorial(int n)  {  if (n == 0)  return 1;  else  return n\*factorial(n-1);  } |

4.1.4 피보나치 수열을 구하는 재귀 함수

- 피보나치(Fibonacci)수열: f[n] = f[n-1] + f[n-2], f[1] = f[2] = 0

|  |
| --- |
| int fibonacci(int n)  {  if(n == 1 || n == 2)  return 1;  else  return fibonacci(n-1) + fibonacci(n-2);  } |

- 재귀 나무(Recursive Tree)를 90도로 돌려서 재귀 호출을 많이 들 그린다.

4.1.5 재귀 호출의 전략

- 문제의 크기를 조금씩 줄여가는 방법

- 문제의 크기를 양분하여 가는 방법

-> 분할해서 통치(divide and conquer)

- 문제의 크기를 자르는 것이 아니라 문제 자체에 접근하여 가는 방법

4.1.6 하노이의 탑

- 브라마 사원의 승려들에 의해 행해졌던 깃이다.

- 하노이의 탑은 세 개의 기둥과 서로 다른 크기인 N개의 원반으로 구성된다.

- 원반들은 세 개의 기둥과 서로 다른 크기인 N개의 원반으로 구성된다.

- 이 원반들은 세 개의 기둥 중의 하나에 반드시 꽂혀 있어야 하며, 자신보다 작은 원반 위에는 그 원반을 놓을 수 없다.

- 문제는 기둥 1에 N개의 원반이 모두 꽂혀있는데, 이 원반들을 기둥 2를 이용하여 기둥 3으로 모두 옮기는 것이다.

- 문제 풀이

1. 기둥 1의 원반을 기둥 3으로 옮긴다.

2. 기둥 1의 원반을 기둥 2로 옮긴다.

3. 기둥 3의 원반을 기둥 2로 옮긴다.

4. 기둥 1의 원반을 기둥 3으로 옮긴다.

5. 기둥 2의 원반을 기둥 1로 옮긴다.

6. 기둥 2의 원반을 기둥 3으로 옮긴다.

7. 기둥 1의 원반을 기둥 3으로 옮긴다.

- 재귀적 표현

1. 기둥 1에서 N-1개의 원반을 기둥 2로 옮긴다.

2. 기둥 1에서 1개의 원반을 기둥 3으로 옮긴다.

3. 기둥 2에서 N-1개의 원반을 기둥 3으로 옮긴다.

|  |
| --- |
| void hanoi(int n, int from, int by, int to)  {  if(n == 1) // 1개의 원반을 움직이는 문제로 줄어들었다.  move(from, to); // from->to;  else  {  hanoi(n-1, from, to, by); // 알고리즘의 1번  move(from, to); // 알고리즘의 2번  hanoi(n-1, by, from, to); // 알고리즘의 3번  }  } |

4.2. 재귀 함수를 비재귀 함수로 바꾸기

- 재귀 호출을 사용하는 함수는 그렇지 않은 함수에 비해 속도가 느리다.

-> 재귀호출이 연속적인 함수 호출로 이루어지기에 함수를 호출하기 위한 준비(오버해드)에 소요되는 시간이 많기 때문

- 재귀 호출을 사용하는 함수는 시스템 다운의 우려가 있다.

-> 재귀 호출이 빠지않고 계속 더해지는 경우 내부 스택 영역을 벗어나 침범해서는 안될 영역을 침범하게 될 수 있다.

따라서 필요한 경우 재귀 함수를 비재귀 함수로 바꾸는 과정이 필요하다.

4.2.1 재귀 호출이 하나인 경우

- factorial() 함수 -> 비재귀 함수

|  |
| --- |
| int iter\_factorial(int n)  {  int f = 1;  while(n > 0)  {  f = f \* n--;  }  return f;  } |

4.2.2 재귀 호출이 둘인 경우1

<유형 1> 뿌리를 먼저 타는 방법 (전위순회; Preorder traverse)

|  |
| --- |
| void preorder\_traverse (node \*t)  {  if(t != tail)  {  visit(t);  preorder\_traverse(t->left);  preorder\_traverse(t->right);  }  } |

|  |
| --- |
| void preorder\_traverse(node \*t)  {  init\_stack();  push(t);  while(!is\_stack\_empty()) // 스텍이 비었으면 끝  {  t = pop(); // 팝하여 t에 대응  if(t != tail) // 종료 조건이 아니면  {  visit(t); // process()  push(t->right); // 변화된 인자 2 푸시  push(t->left); // 변화된 인자 1 푸시  }  }  } |

<유형2> 뿌리를 중간에 타는 방법 (중간순회; inorder traverse)

|  |
| --- |
| void inorder\_traverse(node \*t)  {  if (t != tail)  {  inorder\_traverse(t->left);  visit(t);  inorder\_traverse(t->right);  }  } |

|  |
| --- |
| void inorder\_traverse\_rse(node \*t)  {  int done = 0;  init\_stack();  while (!done)  {  while (t != tail) // 종료 조건  {  push(t); // 인자리스트 푸쉬  t = t->left; // 인자 리스트 변경 1  }  if (!is\_stack\_empty())  {  t = pop(); // 인자 리스트 = pop();  visit(t); // process()  t = t->right; // 인자 리스트 = 변경된 인자 2;  }  else  done = 1;  }  } |

|  |
| --- |
| void nr\_hanoi(int n, int from, int by, int to)  {  int done = 0;  init\_stack(); // 스택을 초기화  while(!done)  {  while(n>1) // 종료 조건이 아니면  {  push(to); // 인자 리스트를 푸시  push(by);  push(from);  push(n);  n--; // 인자 리스트= 변경된 인자  push(to); // to와 by의 교환을 위해 임시로 저장  to = by;  by = pop();  }  move(from, to); // 종료 처리  if(!is\_stack\_empty())  {  n = pop(); // 인자 리스트 = pop();  from = pop();  by = pop();  to = pop();  move(from, to); // process()  n--; // 인자 리스트 = 변경된 인자 2  push(from); // from과 by의 교환을 위해 임시로 저장  from = by;  by = pop();  }  else  done = 1; // 스택이 비면 끝  }  } |

4.2.4 재귀 호출이 둘인 경우3

- 뿌리를 나중에 타는 방법(후위순회; postorder traverse)

|  |
| --- |
| void postorder\_traverse(node \*t)  {  if(t != tail)  {  postorder\_traverse(t->left);  postorder\_traverse(t->right);  visit(t);  }  } |

|  |
| --- |
| void postorder\_traverse\_rse(node \*t)  {  int done = 0;  node \*s;  init\_stack();  while(!done)  {  while(t != till) // 종료 조건이 아닐 동안  {  push(t); // 인자 리스트의 푸시  t = t->left; // 인자 리스트 = 변경된 인자 1  }  while(!is\_stack\_empty())  {  s = t; // 인자 리스트의 복사본 = 인자 리스트  t = pop(); // 인자 리스트 = pop();  if(t->right != tail) // 인자의 변경 2가 종료 조건이 아니면  {  if(t->right != s) // 인자의 변경 2가 인자 리스트의  visit(t); // 복사본과 같으면 precess();  else // 인자의 변경 2가 인자 리스트의 복사본과 다르면?  {  push(t); // 인자 리스트의 푸시  t = t->right; // 인자 리스트 = 변경된 인자 2  break;  }  }  else  visit(t);  }  if(is\_stack\_empty())  done = 1;  }  } |

4.2.5 그 외의 경우

- 재귀 호출이 3회 이상일 경우

- <유형 1>과 유사하게 풀이 할 수 있다..

🡪 4.3.3 참고 recur\_fill(), stack\_fill()

4.2.6 새로운 접근 방법으로 비재귀판을 만드는 경우

|  |
| --- |
| int fibonacci(int n)  {  if(n == 1 || n == 2)  return 1;  return fibonacci(n-1)\*factorial(n-2);  }  int iter\_fibonacci(int n)  {  int r = 0;  int a = 1, b = 1;    if(n == 1 || n == 2)  return 1;    while (n-- > 2)  {  r = a + b; // 값의 교환에 의해 피보나치수 구함  a = b;  b = r;  }  return r;  } |

- Fibonacci() 함수에 비하여 휠씬 빠르고 안정된 성능을 가진다.

4.2.7 주의점

- 상황에 따라 재귀함수가 좋을 수도 있다.

- 속도 향상을 위해 매크로 함수를 이용하는 것이 좋다.

-p.289

4.3 그래픽에서의 활용

- 그래픽 라이브러리에서 각종의 기능이 있으며 재귀적으로 정의되는 경우가 많다.

- 실용성이 거의 없기 때문에 참고만 하기

4.3.1 선 그리기

- 방법

1. 중점에 점을 찍음

2. 새로운 중점에 점을 찍음

3. 이런 작업을 반복하여 선이 그려진다.

\*p.290 그림4-5참고

|  |
| --- |
| void recursive\_line(int x1, int y1, int x2, int y2, int color)  {  if(x1-x2 <= 1 && x1-x2 >= -1 && y1-y2 <= 1 && y1-y2 >= -1)  return; //x1과x2, y1과y2의 차이가 1보다 작아지면 끝  else  {  putpixel((x1+x2)/2, (y1+y2)/2, color);  recursive\_line(x1, y1, (x1 + x2)/2, (y1+y2)/2, color);  recursive\_line(x2, y2, (x1 + x2)/2, (y1+y2)/2, color);  }  } |

- 실용적으로 선을 그리는 함수를 작성하기 위해서는 브레제냄(Bresenham)의 알고리즘을 많이 사용한다.

4.3.2 영역의 내부를 칠하기

- 래스터 그래픽(Raster graphic): 작은 점의 색깔로 그림을 그리는 방식

<영역을 칠하는 알고리즘 (X, Y, 칠하는 색, 경계색)>

1. 현재의 (X,Y)의 점을 읽어 그것이 경계색이거나 이미 채워진 부분이면 끝낸다.
2. 현재의 (X,Y)의 점을 지정된 색으로 찍는다.
3. (X-1, Y)에 대해 영역을 칠하는 알고리즘을 실행한다.
4. (X+1, Y)에 대해 영역을 칠하는 알고리즘을 실행한다.
5. (X, Y-1)에 대해 영역을 칠하는 알고리즘을 실행한다.
6. (X, Y+1)에 대해 영역을 칠하는 알고리즘을 실행한다.

- 처음에 주어지는 좌표(x,y)를 씨앗(Seed)라고한다.

- 경계색이 아니라면 색을 칠하고, 다시 사방으로 씨앗을 퍼트려서 계속 색을 칠한다.

|  |
| --- |
| \*\*알고리즘 |
| void recur\_fill(int x, int y, int color, int border)  {  if(getpixel(x, y) == color || getpixel(x, y) == border)  return;  else  {  putpixel(x, y, color);  recur\_fill(x - 1, y, color, border);  recur\_fill(x + 1, y, color, border);  recur\_fill(x, y - 1, color, border);  recur\_fill(x, y + 1, color, border);  }  } |

|  |
| --- |
| \*\* 알고리즘의 비재귀판 |
| void stack\_fill(int x, int y, int color, int border)  {  init\_stack();  push(y);  push(x);  while (!is\_stack\_empty())  {  x = pop();  y = pop();  if(getpixel(x, y) == color || getpixel(x,y) == border)  continue;  putpixel(x, t, color);  push(y + 1);  push(x);  push(y - 1);  push(x);  push(y);  push(x + 1);  push(y);  push(x - 1);  }  } |

- 실행너무 잘되요 재밌어욧!

4.4 프랙탈 그래픽

- 프랙탈(Fractal): 어떤 수학적공식이나 물리형상을 그림으로써 나타냄

- 규칙성이 없을 것 같은 산의 모양과 나무의 모양의 규칙을 찾아내어서 수식으로 화면상에 연출함

4.4.1 양탄자 모양

- 시어핀스키의 양탄자라는 정통적인 모양

- 사각형의 모양에서 중앙의 사각형을 다른 색으로 칠하는 과정을 반복함

<시어핀스크의 양탄자 알고리즘 (x, y, r)>

1. r이 0이 되면 끝이다.(종료 조건)

2. (x-2\*r, y+2\*r, r/3)에 대해 알고리즘을 실행한다.

3. (x-2\*r, y , r/3)에 대해 알고리즘을 실행한다.

4. (x+2\*r, y-2\*r, r/3)에 대해 알고리즘을 실행한다.

5. (x , y+2\*r, r/3)에 대해 알고리즘을 실행한다.

6. (x , y-2\*r, r/3)에 대해 알고리즘을 실행한다.

7. (x+2\*r, y+2\*r, r/3)에 대해 알고리즘을 실행한다.

8. (x+2\*r, y , r/3)에 대해 알고리즘을 실행한다.

9. (x+2\*r, y-2\*r, r/3)에 대해 알고리즘을 실행한다.

10. (x-r/3, y-r/3, x+r/3, y+r/3)에 크기의 사각형을 덜어낸다.

|  |
| --- |
| \*\*시어핀스키의 양탄자 알고리즘 |
| void carpet1(int x, int y, int r)  {  if(r > 0)  {  carpet1(x - 2\*r, y + 2\*r, r/3); // 좌측 하단  carpet1(x - 2\*r, y , r/3); // 좌측  carpet1(x - 2\*r, y - 2\*r, r/3); // 좌측 상단  carpet1(x , y + 2\*r, r/3); // 하단  carpet1(x , y - 2\*r, r/3); // 상단  carpet1(x + 2\*r, y + 2\*r, r/3); // 우측 하단  carpet1(x + 2\*r, y , r/3); // 우측  carpet1(x + 2\*r, y - 2\*r, r/3); // 우측 상단  box(x, y, r); // 사각형 그림  }  } |

|  |
| --- |
| \*\*사각형 그림 함수 |
| void box(int x, int y, int r)  {  int i;  setcolor(WHITE);  for(i = y-r/3; i <= y+r/3; i++)  line(x-r/3, i, x+r/3, i);  } |

4.4.2 프랙탈 나무

- 나무의 가지가 벌어지는 모양을 수학적으로 분석을 하여 재귀적으로 그리는 방법

<프랙탈 나무를 그리는 알고리즘(order, length, angle)>

1. 주어진 length와 angle을 이용하여 이동할 상대 좌표를 구한다.

2. 구한 상대 좌표로 선을 그린다.(제어점이 상대 좌표만큼 이동한다.)

3. order가 0이면 6으로 간다.(종료 조건)

4. (order-1, length\*SCALE, angle+TURN)에 대해 프랙탈 나무 알고리즘을 실행한다.

5. (order-1, length\*SCALE, angle+TURN)에 대해 프랙탈 나무 알고리즘을 실행한다

6. 제어점을 1의 상태로 되돌린다.

|  |
| --- |
| \*\*프랙탈 나무 알고리즘 |
|  |

// 중요한건 아닌거 같아서 그만 적음 자세한 내용은 p.300쪽 언저리

4.5 파일 찾기 프로그램 RFF

- 재귀 호출을 이용한다.

4.5.1 findfirst(), findnext()의 사용법

4.6 결론

- 재귀 호출은 알고리즘의 구현에 필수적이다.

- 재귀적인 함수를 비재귀판으로 만드는 방법을 통해 최적화의 기법을 배웠다.

<이장을 통해 배운 것>

1. 재귀적인 문제를 재귀적인 알고리즘으로 해결

2. 재귀적인 함수 작성

3. 재귀 함수의 실행을 분석

4. 재귀 함수를 비재귀함수로 변환

5. 정렬 알고리즘

5.1 개요

- 정렬(Sort): 임의의 순서대로 배열되어 있는 자료의 집합을 일정한 순서대로 재배열 한 것

- 퀵 정렬(Quick Sort): 가장 빠른 정렬 알고리즘

- 이장에서는 다양한 정렬 알고리즘에 대해서 그 원리와 c로의 구현, 그리고 분석을 한다

5.1.1 정렬의 방법론

- 정렬 알고리즘은 판단과 교환을 어떻게 적절히 조합하는가에 대한 방법론이다.

- 판단(decision): 저 카드보다 큰지 작은지를 가려낸다.

- 교환(exchange): 이 카드가 저 카드보다 크면 두 카드를 바꾼다.

- 정렬을 하는 대상을 파일(file)이라고 하면, 이 파일은 정보의 단위인 레코드(record)의 나열이다.

- 다음은 인사 기록 카드(레코드)를 C로 표현한것이다.

|  |
| --- |
| struct person  {  int code; // 사번  int sex; // 성별  int age; // 나이  char name[9]; // 성명  char state[11]; // 직책  }; |

- 정렬에는 키(key)가 필요하며 키의 값을 비교하여 정렬이 이루어짐

- 오름차순(ascending order): 키 값을 비교하여 작은 값을 앞에 큰 값을 뒤에 두는 방법

- 내림차순(descending order): 그 반대

- ‘비교’의 대상이 되는 것은 레코드중에서도 키값이며, ‘교환’의 대상이 되는 것은 레코드 자체가 된다.

5.1.2 정렬 알고리즘의 다양성

정렬 알고리즘은 참으로 다양하다.

|  |  |
| --- | --- |
| 간단한 알고리즘 | 1. 선택 정렬  2. 삽입 정렬  3. 거품 정렬  4. 셀 정렬 |
| 복잡한 알고리즘 | 1. 퀵 정렬  2. 기수 정렬  3. 힙 정렬  4. 병합 정렬 |

- 간단한 알고리즘은 안정되어 있고 구현이 쉽다.

- 복잡한 알고리즘에 비해서 정렬 속도가 느리다.

- 복잡한 알고리즘은 구현에 어려움이 있거나 부차적인 메모가 필요하다.

- 단순한 알고리즘에 비해서 정렬 속도가 빠르다.

정렬의 대상이 되는 레코드들의 배열이 컴퓨터 메모리 내부에 있는가 아니면 디스크나 테이프 장치와 같은 컴퓨터의 외부에 있는가에 따라서 내부 정렬(Internal Sort)과 외부 정렬(External Sort)로 나눌 수 있다.

내부 정렬

장점

- 컴퓨터 메모리의 바른 비교와 교환을 이용하기 때문에 속도가 빠르다

단점

- 모든 자료를 컴퓨터의 메인 메모리로 옮겨 놓아야 한다는 부담이 있다.

외부 정렬

- 디스크나 테이프상에서 직접 정렬을 행하기 때문에 속도는 느리지만 메인 메모리를 자료들이 차지하지는 않는다.

- 파일을 잘라서 메모리에 차례로 읽어들여 내부 정렬한 다음 전체로 병합(Merge)하는 방법을 사용한다.

- 레코드의 전체를 교환하느냐, 레코드를 가르키는 포인터를 가르키느냐에 따라 직접 정렬(Direct Sort)와 간접 정렬(Indirect Sort)로 나누기도 한다.

- 직접 정렬의 경우 부가적인 메모리가 필요없어 메모리는 절야기 되나 교환 작업시 복사를 해야하는 자료의 크기가 커서 속도가 느려진다.

- 간접 정렬의 경우 레코드를 가리키는 포인터와 배열이 필요해서 메모리의 소요가 있지만 실제로 레코드의 교환 작업은 일어나지 않아 속도 향상을 기대할 수 있다.

정렬의 기본 방법에 따른 분류

- 삽입(Insertion): 삽입 정렬과 쉘 정렬

- 교환(Exchange): 거품 정렬과 퀵 정렬, 기수 교환 정렬

- 선택(Selection): 선택 정렬, 힙 정렬

- 세기(Counting): 분포수세기, 직접 기수 정렬

5.1.3 안정성(Stability)

- 안정성이란 같은 내용을 가지는 키값의 배열이 정렬 후에도 상대적 순서 유지 유무

- 다중키(Multil)정렬: 여러가지 키로 하는 것

- 일차키(Frimary key): 우선순위가 가장 놓은 키

- 2차키(Secondary key): 다음로 우선순위가 높은 키

5.1.4 일반화된 정렬 함수

5.2 선택 정렬(Selection Sort)

- 선택 정렬은 가장 간단한 정렬 알고리즘이며 실생활에서 가장 많이 사용하는 알고리즘이다.

5.2.1 선택 정렬 알고리즘

- 다음은 선택 정렬을 하는 알고리즘이다.

1. i=0

2. i가 n-2가 되면 끝낸다.

3. 배열의 i향부터 n-1항까지중 최소값을 찾아서 그 항을 min에 저장한다.

4. 배열의 i항과 min항을 교환한다.

5. i를 하나 증가시키고 2로 돌아간다.

5.2.2 select\_sort()함수

- select\_sort()함수는 정수형의 배열과 배열 요소의 개수를 인자로 받아 선택 정렬을 한다.

|  |
| --- |
| void select\_sort(int a[], int n)  {  int min; // 최소값을 저장  int minindex; // 최소값의 인덱스를 저장  int i, j;    for(i = 0; i < n-1; i++)  {  minindex = i; // 최소값의 초기 설정  min = a[i];  for(j = i + 1; j < n; j++) // i 이후로 최소값을 찾음  {  if(min > a[j]) // 더 작은 값이 나타나면 바꿈  {  min = a[j];  minindex = j;  }  }  a[minindex] = a[i]; // i항과 min항을 교환  a[i] == min;  }  } |

5.2.3 선택 정렬의 과정

- 선택 정렬의 비교 횟수는 n의 2승/2정도로 많지만 교환 횟수는 많아야 n번이다.

- 선택 정렬 알고리즘은 이중 루프가 있어야 하기 때문에 N수가 커지면 실행시간이 제곱으로 늘어난다.

5.2.4 안정성 문제

- 안정성(stablilty)란 같은 키의 상대적 순서가 정렬 후에도 변함이 없음을 뜻한다.

- 하지만 선택 정렬은 안정성이 없다.

- 선택 정렬과 같이 인접한 배열 요소를 교환하지 않고 뚝 떨어진 요소를 교환하는 경우에는 안정성이 없는 경우가 많다.

5.2.5 선택 정렬 알고리즘의 실행 시간 분석

- n의 4재곱

5.2.6 최적화 기법

- 선택 정렬은 실행 시간이 N의 제곱에 비례하기 때문에 큰 자료에 대해서는 적용이 힘들지만 알고리즘이 간단하고 교환 횟수가 적어 큰 레코드의 직접 정렬에는 호용이 있다.

- 알고리즘 자체는 단순해서 더 이상 손댈 것이 없다.

- select\_sort()함수를 최적화해야 한다.

-> 내부 루프를 기계어(machine language)로 작성해보는 것이다.

5.2.7 일반적인 함수

- base: 배열의 선두 번지

- nelem: 자료의 개수

- width: 레코드의 크기

- fcmp: 키값을 비교하는 함수

|  |
| --- |
| \*\*정수 배열의 비교 함수 |
| int intcmp(const void \*a, const void \*b)  {  return (\*(int\*)a - \*(int\*)b);  } |

- 즉 인자로 주어진 a와 b에 대해서 a>b이면 0보다 큰 값을 a==b이면 0을 a<b이면 0보다 작은 값을 리턴하면 된다.

|  |
| --- |
| 일반형 선택 정렬 함수 select\_sort() |
| void select\_sort(void \*base, size\_t nelem, size\_t width, int(\*fcmp)(const void\*, const void\*))  {  void \*min;  int minindex;  int i, j;    min = malloc(width); // min에 공간 할당  for(i = 0; i < nelem - 1; i++)  {  minindex = i;  memcpy(min, (char\*)base + i\*width, width); // min = a[i]  for(j = i + 1; j < nelem; j++)  {  if(fcmp(min, (char\*)base + j\*width) > 0) // min > a[j]  {  memcpy(min, (char\*)base + j\*width, width); // min = a[j]  minindex = j;  }  }  memcpy((char\*)base + minindex\*width, (char\*)base + i\*width, width);  // a[minindex] = a[i]  memcpy((char\*)base + i\*width, min width);  // a[i] = min  }  free(min); // min 공간의 해제  } |

5.3 삽입 정렬(Insertion Sort)

- 삽입 정렬은 성택 정렬과 함께 가장 많이 사용되는 정렬 방법이다.

- 삽입 정렬은 적은 비교와 많은 교환이 특징이다. -> 선택정렬과 반대의 특징

5.3.1 삽입 정렬의 전략

- 삽입 정렬은 이미 정렬이 된 부분에 새로운 키를 적절한 장소에 삽입하는 동작을 반복적으로 하는 정렬 방법이다.

< 삽입 정렬 알고리즘 >

1. i=1;

2. j=I;

3. a[j-1] > a[i]이고 j>0인 동안

3.1 a[j] = a[j-1] -🡪 a[i]를 삽입할 공간을 만든다.

3.2 j를 하나 감소한다.

4. a[j] = a[i] -🡪 빈 공간에 a[i]를 삽입한다.

5. i를 하나 증가하고 2로 돌아간다.

- 위 알고리즘에 의하면 배열에서 I 향 이하는 이미 정렬된 상태가 되며 i향을 적절한 위취에 삽입하는 동작은 j 변수를 통해서 구현한다.

5.3.2 삽입 정렬의 실제

- 삽입정렬은 안정성이 있다.

\*p.330

5.3.3 c로 구현한 삽입 정렬

|  |
| --- |
| \*\*삽입 정렬 알고리즘 |
| void insert\_sort(int a[], int n)  {  int i, j, t;  for(i = 1; i < n; i++)  {  t = a[i];  j = i;  while(a[j-1]> t && j > 0) // 삽입될 곳을 찾음  {  a[j] = a[j-1]; // 뒤로 옮김  j--;  }  a[j] = t; // 삽입함  }  } |

5.3.4 삽입 정렬의 분석

1. 삽입 정렬의 이미 정렬된 배열일 경우 매우 빠른 속도를 가진다.

- 즉 이미 정렬된 경우에는 삽입 정렬의 실행 시간은 n에 비례하는 성형적인 특징을 가진다.

2. 삽입 정렬은 역순의 경우 최악의 경우가 된다.

-> 이동과 비교의 횟수는 N\*(N-1)/2가 되어서 선택 정렬보다 속도가 떨어진다.

\*선택 정렬은 N\*(N-1)/2번 비교하지만 교환은 N번만 한다.

3. 난수 배열인 경우에는 선택 정렬보다는 횔씬 효율이 좋다.

4. 대충 정렬된 배열의 경우 삽입 정렬은 속도가 굉장히 빠르다.

->비교와 교환의 횟수가 작아지므로 빨리질 수 밖에 없다.

결론: 삽입 정렬은 입력 자료에 굉장히 민감하다.

-> 입력 자료의 정렬된 정도에 따라 효율이 좋을 수도 있고 안 좋을 수도 있다.

-> 작은 크기의 대충 정렬된 배열에 대해서는 제일의 선택이된다.

하지만 비교횟수는 적고 교환 횟수가 많아 큰 레코드를 정렬할 때에는 부적절하다.

-> 삽입 정렬은 작은 레코드의 배열에 사용하면 뛰어난 성능을 발휘할 것이다.

마지막으로 삽입 정렬은 0(N의 2승)의 성능을 나타낸다.

5.3.5 삽입 정렬의 최적화

- 보초를 사용하여 내부 루프를 간단히 만든다.

-원리는 간단. A[0]에 -1를 집어 넣어 j>0 비교문을 없앨 수 있다.

|  |
| --- |
| \*\*보초(sentinal)를 사용한 삽입 정렬 알고리즘 |
| void insert\_sort1(int a[], int n)  { // 보초를 이용한 삽입 정렬, a[0]에는 가장 작은 값이 들어있어야 함  int i, j, t;  for (i = 2; i <= n; i++)  {  t = a[i];  j = i;  while (a[j-i] > t) // j>0이 없어졌다.  {  a[j] = a[j-1];  j--;  }  a[j] = t;  }  } |

- 하지만 많은 경우가 키값의 최소값을 정하지 못하기 때문에 보초기법의 사용은 신중해야 한다.

- N의 크기가 클수록 보초기법을 사용한 보람을 느낄수 있다.

나중에 배우는 이분 검색을 이용하면 삽입 정렬을 개선할 수 있다.

5.3.6 간접 정렬

삽입 정렬에서 간접 정렬(Indirect Sort)를 고려하는 이유

- 삽입 정렬이 비교 횟수는 적지만 교환 횟수가 많기 때문

- 간접 정렬은 레코드의 배열은 전혀 손대지 않고 따로 만든 인덱스(index)의 배열을 조작하는 방법이다.

|  |
| --- |
| \*\*간접 정렬을 하는 indirect\_insert\_sort() |
| void indirect\_insert\_sort(int a[], int index[], int n)  {  int i, j;  int t;  for(i = 0; i < n; i++) // 인덱스의 배열을 초기화  index[i] = i;  for(i = 1; i < n; i++) // 삽입 정렬  {  t = index[i];  j = i;  while(a[index[j-1]] > a[t] && j > 0) // 삽입될 곳을 찾음  {  index[j] = index[j-1]; // 인덱스의 배열을 조정  j--;  }  index[j] = t; // 삽입함  }  } |

- 간접 정렬을 하고나면 실제로 a배열에는 아무 변동이 없다.

-> 그러나 그 정렬된 순서는 index배열에 저장되어 있다.

-> 만약 첫번째 레코드를 알고 싶을 경우 a[index[0]]

- 이것으로 부족하고 실제로 a배열을 정렬해야 한다면 rearrange()함수를 사용하면 된다.

|  |
| --- |
| \*\*rearrange()함수 |
| void rearrange(int a[], int index[], int n)  {  int \*p;  int i;  p = (int\*)malloc(sizeof(int)\*n);  for(i=0; i<n; i++)  p[i] = a[index[i]];  for(i=0; i<nl i++)  a[i] = p[i];  free(p);  } |

- 실제로 a배열을 정렬하기 위해서 따로 p라는 배열을 생성하고 여기에 index를 이용하여 정렬된 순서대로 레코드들을 저장한다.

- 실제의 사용에 있어서는 메모리를 사용하는 내부 정렬의 경우에는 큰 레코드라 할지라도 메모리의 쓰고 읽는 속도가 워낙 빠르기 때문에 굳이 간접 정렬을 쓸 필요가 없다.

-> 하지만 외부 정렬의 경우는 간접 정렬이 필수적인 경우가 있다.

- 인덱스 기법: 정렬할 키와 자료 파일에서 인덱스만을 따로 뽑아 새로운 인덱스파을을 만든다.

- 인덱스 파일만 정렬하고 검색하여서 매우 빠른 속도로 자료들을 처리한다.

5.3.7 일반화된 함수 작성

- 다음은 void 포인터와 함수 포인터를 이용하여 일반화 된 정렬 함수를 만들수잇다.

- base: 배열의 선두 번지

- nelem: 자료의 개수

- width: 레코드의 크기

- fcmp: 키값의 비교를 하는 함수

|  |
| --- |
| \*\*insert\_sort()함수 |
| void insert\_sort(void\*base, size\_t nelem, size\_t width, int(\*fcmp)(const void\*, const void\*))  {  int i, j;  void \*t;  t = malloc(width);  for(i = 1; i < melem; i++)  {  memcpy(t, (char\*)base + i\*width, width);  j = i;  while(fcmp((char\*)base + (j-1)\*width, t) > 0 && j > 0)  {  memcpy((char\*)base + j\*width, (char\*)base + (j-1)\*width, width);  j--;  }  memcpy((char\*)base + j\*width, (char\*)base + (j-1)\*width, width);  j--;  }  memcpy((char\*)base + j\*width, t, width);  } |

5.4 거품 정렬(Bubble Sort)

- 거품 정렬은 배열의 인접 요소(adjacent element)를 비교하여 교환하는 모양이 마치 거품이 보글 거리는 모양이라고 해서 붙여진 이름이다.

- 거품 정렬은 본서에서 보이는 정렬 알고리즘 중에서 가장 느리고 형편없는 알고리즘이다.

5.4.1 거품 정렬의 전략

- 거품 정렬은 인접한 배열의 요소를 비교 교환하여 전체적으로는 대충 정렬을 하면서 최대값을 배열의 제일 뒤로 보내는 것을 반복한다.

<거품 정렬 알고리즘>

1. I = 0;

2. i 가 n-1이 되면 끝낸다.

3. j = 1

4. j가 n-i이 되면 7로 간다.

5. a[j-1] > a[j]이면 두 값을 교환한다.

6. j를 하나 증가시키고 4로 돌아간다.

7. i를 하나 증가시키고 2로 돌아간다.

5.4.2 거품 정렬의 실제

- 최대값이 뒤로 옮긴다.

- 안정성이 있다.

5.4.3 C로 구현한 거품 정렬

- 거품 정렬을 c로 구현해보자

|  |
| --- |
| \*\* 거품 정렬 알고리즘 |
| void bubble\_sert(int a[], int n)  {  int i, j, t;  for(i = 0; i<n-1; i++)  {  for(j = 1; j < n-i; j++)  {  if(a[j-1] > a[j]) // 인접 요소의 비교  {  t = a[j-1]; // 교환  a[j-1] = a[j];  a[j] = t;  }  }  }  } |

5.4.4 거품 정렬의 분석

- 거품 정렬은 최대값을 찾는 것과 대충 정렬을 하는 두가지 일을 하기 때문에 실행 시간이 굉장히 느리다.

\*p.341 그림

1. 거품 정렬의 최선의 경우로 배열이 임 정렬된 배열일 경우

-> N\*(N-1)/2번의 비교를 하며 교환은 없다.

- 배열을 참조하는 것보다 일반 변수를 참조하는 것이 효율적이다.

2. 거품 정렬의 최악의 경우로 역순 배열일 경우

-> 비교와 교환의 횟수가 공히 N\*(N-1)/2번이다.

- 거품 정렬은 모든 경우에 있어 고르게 속도가 느리다. 그래서 실제 프로그래밍에서 거품 정렬은 거의 사용이 배재된다.

5.4.5 거품 정렬의 개선

- 다음은 정렬되어 있는지 확인하는 is\_sorted함수이다.

|  |
| --- |
| \*\*is\_sorted() |
| int is\_sorted(int a[], int n)  {  int i, s;  s = 1;  for(i = 1; i < n; i++)  if(a[i-1] > a[i]) s = 0;  return s;  } |

- 위 함수를 감안하여 거품 정렬을 개선할 수 있다.

|  |
| --- |
| \*\*개선한 버블 정렬 알고리즘 |
| void bubble\_sert1(int a[], int n)  {  int i, j, t, s;  for(i = 0; i<n-1; i++)  {  s = 1;  for(j = 1; j < n-i; j++)  {  if(a[j-1] > a[j]) // 인접 요소의 비교  {  t = a[j-1]; // 교환  a[j-1] = a[j];  a[j] = t;  s = 0;  }  }  if (s==1)break;  }  } |

- 정렬된 배열에는 시간이 많이 단축됨

- 나머지 들은 전부 느려짐

- if문의 추가로 얼마나 실행 시간에 영향을 주는지 알아두기

거품 정렬의 변형: 쉐이커 정렬(Shaker sort)

- 거품 정렬이 왼쪽에서 오른 쪽 방향으로 인접한 요소들을 비교하여 최대값을 우측으로 보내는 반면,

1. 쉐이커 정렬은 말 그대로 한번은 왼쪽에서 오른쪽으로 가면서 최대값을 우측으로 보내고,

2. 한번은 오른쪽에서 왼쪽으로 가면서 최소값을 좌측으로 보내는 것을 반복한다.

- 쉐이커 정렬은 거품 정렬보다 효율이 더 좋다.

5.4.6 일반화된 함수 작성

|  |
| --- |
| **void bubble\_sort\_pre(void \*base, size\_t nelem, size\_t width, int (\*fcmp)(const void\*, const void\*))**  **{**  **int i, j, s;**  **void \*t;**  **t = malloc(width);**  **for(i = 0; i < nelem-1; j++)**  **{**  **s = 1;**  **for(j = 1; j < nelem-i; j++)**  **{**  **if(fcmp((char\*)base + (j-1)\*width, (char\*)base + j\*width) > 0)**  **{**  **memcpy(t, (char\*)base + (j-1)\*width, width);**  **memcpy((char\*)base + (j-1)\*width, (char\*)base + j\*width, width);**  **memcpy((char\*)base + j\*width, t, width);**  **s = 0;**  **}**  **}**  **if (s == 1) break;**  **}**  **free(t);**  **}** |

5.5 쉘 정렬(Shell Sort)

- 삽입 정렬은 이미 정렬된 배열이나 대충 정렬된 배열에 대해서는 매우 뛰어난 성능을 보이지만 그렇지 않은 경우에는 속도가 매우 느리다.

- 쉘 정렬은 이 같은 문제점을 해결하기 위해서 h만큼의 간격으로 떨어진 레코드를 삽입정렬하는 방법이다.

5.5.1 쉘 정렬의 전략

- 쉘 정렬은 삽입 정렬과 다르게 뚝 떨어진 요소들을 교환한다.

- 쉘 정렬의 기본 개념은 정렬할 배열에서 h간격의 요소들만 뽑아서 삽입 정렬하는 것

<쉘 정렬 알고리즘>

1. h의 초기값을 구한다.

2. h가 1보다 작으면 끝낸다.

3. I = 0

4. i가 h보다 크거나 같으면 7로 간다.

5. (h간격 + i) 요소들에 대해서 삽입 정렬을 한다.

6. i를 하나 증가시키고 4로 간다.

7. h의 다음 값을 구하고 2로 간다.

5.5.2 쉘 정령의 실제

- h를 처음에 10으로 잡고 계속 2로 나누어 가면서 h가 1일 때까지 정렬하는 방법

- 안정성은 없는듯

5.5.3 C로 구현한 쉘 정렬

- 다음은 shell\_sort()함수의 쉘 정렬 알고리즘을 c로 표현한 것이다.

|  |
| --- |
| \*\* 쉘 정렬 알고리즘 |
| void shell\_sort(int a[], int n)  {  int i, j, k, h, v;  for(h = n/2; h > 0; h /= 2) // h <- h/2  {  for(i = 0; i < h; i++) // 변이  {  for(j = i+h; j < n; j += h) // j는 삽입 정렬할 요소를 지정  {  v = a[j];  k = j;  while(k > h-1 && a[k-h] > v) // 삽입할 곳을 찾음  {  a[k] = a[k-h]; // 이동  k -= h;  }  a[k] = v; // 삽입  }  }  }  } |

- h가 들어간 것과 4중 루프가 된 것을 제외하고는 삽입 정렬과 비슷하다

5.5.4 쉘 정렬의 분석

- 쉘 정렬은 h가 점점 작아지면서 차츰 차츰 점진적으로 배열이 정렬되어간다.

- 삽입 정렬을 c로 구현한 shell\_sort()함수는 4중 루프로 구성되어 있어 속도가 메우 느릴 것 같지만 상당히 성능이 좋은 알고리즘이다.

- 쉘 정렬은 h를 정하는 방법에 따라 알고리즘의 성능이 달라진다.

- 일반적으로O(N(logN)의 2승)이나 O(N의 1.24승)정도의 성능을 가진다.

5.5.5 쉘 정렬의 계선

- 쉘 정렬의 계선 방안으로 h를 잘 설정하는 것이다.

|  |
| --- |
| \*\* Robert Sedgewick |
| void shell\_sort1(int a[], int n)  { // h = 3\*h + 1  int i, j, k, h, v;  for(h = 1; h < n; h = 3\*h+1); // n보다 작은 h의 최대값을 찾는다.  for(h /= 3; h > 0; h /= 3)  {  for(i = 0; h > 0; h /= 3)  {  for(i = 0; i < h; i++)  {  v = a[j];  k = j;  while(k > h-1 && a[k-h] > h)  {  a[k] = a[k-h];  k -= h;  }  a[k] = v;  }  }  }  } |

5.5.6 일반화된 함수 작성

- 가장 빠르다고 알려진 h = 3\*h+1수열을 이용하여 일반화한 쉘 정렬 함수

|  |
| --- |
| \*\* |
| void shell\_sort(void \*base, size\_t nelem, size\_t width, int (\*fcmp)(const void\*, sonst void\*))  {  int i, j, k, h;  void \*v;  v = malloc(width);  for (h = 1; h < nelem; h = 3\*h+1);  for (h /= 3; h > 0; h /= 3)  {  for(i = 0; i < h; i++)  {  for(j = i+h; j < nelem; j += h)  {  memcpy(v, (char\*)base + j\*width, width);  k = j;  while(k > h-1 && fcmp((char\*)base + (k-h)\*width, v) > 0)  {  memcpy((char\*)base+k\*width,(char\*)base+(k-h)\*width,width);  k -= h;  }  memcpy((char\*)base+k\*width, v, width);  }  }  }  free(v);  } |

5.6 분포수세기(Distribution Counting)

- 분포수세기는 같은 키가 많이 있는 배열에 한해 적용할 수 있는 정렬 알고리즘이다.

- 분포수세기는 특정 키값이 출현하는 빈도를 저장하여 누적분포를 이용하여 간단하게 정렬하는 방법

5.6.1 분포수세기의 전략

<분포수세기 알고리즘 정리>

1. count[] 배열을 0으로 초기화

2. a[] 배열의 키의 빈도를 계산하여 그 빈도를 count[]에 저장

3. count[] 배열을 누적분포로 변환

4. a[] 배열을 뒤에서부터 읽어서 b[--count[a[i]]]에 저장.

5. b[] 배열을 a[] 배열로 복사

5.6.2 분포수세기의 실제

- p.353

5.6.3 C로 구현한 분포수세기

- 알고리즘 구현

|  |
| --- |
| \*\* |
| void dist\_count(int a[], int n, int m)  {  int i;  int \*b, \*count;  b = (int\*)malloc(sizeof(int)\*n); // b 배열 생성  count = (int\*)malloc(sizeof(int)\*(m+1)); // count 배열 생성  for(i = 0; i <= m; i++) // count 배열을 초기화  count[i] = 0;  for(i = 0; i < n; i++) // 빈도수 계산  count[a[i]]++;  for(i = 2; i <= m; i++) // 누적 분포 계산  count[i] = count[i-1] + count[i];  for(i = n-1; i >= 0; i--) // 뒤에서부터 읽어 b에 저장  b[--count[a[i]]] = a[i];  for(i = 0; i < n; i++) // b에서 a로 복제  a[i] = b[i];  free(b);  free(count);  } |

- 일반화된 정렬 함수를 작성하는 것은 매우 힘이 든다.

-> 배열에 따라 가지는 값의 범위가 모두 다르기 때문이다.

5.6.4 분포수세기의 분석

- 분포수세기는 본격적인 정렬 알고리즘이라고 하기에는 한정된 용도에서만 적합하다.

- 우선속도가 약 2N번의 비교와 1번의 전체 복사가 있는 정도여서 속도가 빠르다.

하지만 키 값의 분포를 저장하는 count배열과 작업 결과를 입시로 저장할 b배열을 생성해야 하므로 메모리의 소모가 너무 크다.

- 분포수세기 알고리즘은 기수 정렬에서 사용이 되어 강력한 기능을 발휘한다.

5.7 퀵 정렬(Quick Sort)

- 퀵 정렬은 1960년에 C.A.R 이라는 분에 의해서 처음으로 고안된 가장 널리 사요되는 정렬 방법 중 하나이다.

- 퀵 정렬은 아주 빠른 속도를 나타낼뿐만 아니라 원리도 간단해서 많은 응용 분야에서 사용된다.

5.7.1 퀵 정렬의 전략

- 연속적인 분할(Partition)에 의해서 정렬한다.

-> 분할의 의미는 단순히 입력 배열을 나누는 것이 아니라 축(Pivot)값을 중심으로 왼쪽은 이 축값보다 작은 값으로 오른 쪽은 모두 이 축값보다 큰 값으로 배열시키는 것이다.

<퀵 정렬 알고리즘(a, N)>

1. 만약 N>1이면.

1.1 N 크기의 a배열을 분할(partition)하여 축값의 위치를 mid로 넘긴다.

1.2 퀵 정렬 알고리즘(a, mid)

1.3 퀵 정렬 알고리즘(a+mid+1, N-mid-1)

- 재귀호출

5.7.2 퀵 정렬의 실제

- 퀵 정렬 알고리즘은 재귀적이기 때문에 조심스럽게 그 과정을 추적해야 한다.

- 쉘 정렬처럼 멀리 떨어져 있는 요소끼리 교환을 하기 때문에 빨리 정렬된 상태로 변화된다.

- 교환과 비교 횟수가 매우 적다.

- 안정성이 없다.

5.7.3 C로 구현한 퀵 정렬

- 퀵 정렬은 C로 구현할 때에는 분할에 신경써야 한다.

- 다음은 quick\_sort()함수이다.

|  |
| --- |
| \*\* |
| void quick\_sort(int a[], int n)  {  int v, t;  int i, j;  if(n > 1) // 재귀호출 종료 조건  {  v = a[n-1]; // v는 축값  i = -1; // i는 왼쪽에서부터 검색할 위치 저장  j = n-1; // j는 오른쪽에서부터 검색할 위치 저장  while(1)  {  while(a[++i] < v); //왼쪽에서부터 축값보다 큰 값이 있나?  while(a[--j] > v); //오른쪽서부터 축값보다 작은 값 있나?  if(i >= j) break; // i와 j의 위치가 뒤바뀌었으면 분할 끝  t = a[i]; // 아니면 두 값을 바꿈  a[i] = a[j];  a[j] = t;  }  t = a[i]; // 축값과 축갑의 취티에 있는 값을 바꿈  a[i] = a[n-1];  a[n-1] = t;  quick\_sort(a, i); // 왼쪽 소구간에 대해 퀵 정렬  quick\_sort(a+i+1, n-i-1); // 오른쪽 소구간에 대해 퀵 정렬  }  } |

5.7.4 퀵 정렬의 분석

- 핵심은 한번 수행시 배열 전체를 정렬된 상태로 가깝게 만든다.

-> 즉 쉘 정렬과 마찬가지로 점진적인 정렬 형태를 보인다.

- 난수배열이나 반쯤 정렬된 배열에 대해서는 빠른 속도를 보인다.

-> 하지만 이미 정렬된 배열이나 역순 배열에 대해서는 너무도 엄청나게 성능이 떨어진다.

-> 조금만 생각해보면 그 이유를 알수 있음

- 퀵 정렬의 이상적인 경우는 분할이 정확하게 구간을 양분하는 것이다.

-> 이경우 log(작은2)2N이 되며 가장 빠른 속도를 나타낸다.

- 퀵 정렬은 일반적인 경우 NlogN의 평균 실행 시간을 가지는 알고리즘으로 N이 크게 늘어나더라도 실행 시간은 그에 비례하는 정도로 밖에 늘지 않는다.

- 퀵 정렬이 이렇게 속도가 빠른 이면에는 메모리의 희생이 있다.

-> 빈번한 재귀 호출로 인해서 내부 스택이 사용되기 때문

5.7.5 퀵 정렬의 개선 1: 비재귀판

- 퀵 정렬의 장점에도 불구하고 문제가 있다.

-> 내부 스택의 오버플로

- 한번의 재귀호출로 복귀 주소 2바이트, n 2바이트, a 2바이트해서 6바이트의 스택이 사용된다.

2000개 정도의 정수를 퀵 정렬시 최악의 경우(역순 배열or 정렬된 배열)N만큼 재귀가 깊어지므로 20000\*6=120000바이트 약 120kbyte 가량이 소모되어 스택은 넘치게 된다.

- 이런 문제를 해결하는 방안으로 재귀호출을 없애고 직접 스택을 만들어 쓰는 것이다.

|  |
| --- |
| \*\* 비재귀판 퀵 정렬 알고리즘 |
| void quick\_sort1(int a[], int n)  { // 재귀호출을 없앤 퀵 정렬  int v,t;  int i, j;  int l, r; // 스택에 저장할 구간의 정보  init\_stack();  l = 0;  r = n-1;  push(r);  push(l);  while(!is\_stack\_empty())  {  l = pop();  r = pop();  if (r-l+1 > 1) // 종료 조건 r-l+1은 구간의 길이  {  v = a[r]; // v는 축의 값  i = l-1;  j = r;  while(1) // 분할  {  while(a[++i] < v);  while(a[--j] > v);  if(i >= j) break;  t = a[i];  a[i] = a[j];  a[j] = t;  }  t = a[i];  a[i] = a[r];  a[r] = t;  push(r); // 두번째 재귀함수  push(i+1);  push(i-1); // 첫번째 재귀함수  push(l);  }  }  } |

- 기존 알고리즘보다 약간 빠르다.

- 오버헤드형상으로 속도가 시원하게 오르지 않는다.

- 또한 최악의 경우인(정렬된 배열과 역순배열)의 경우 40배 가량 더 느리다.

-> 이 문제를 근본적으로 해결해야한다.

5.7.6 퀵 정렬의 개선 2: 난수 분할

- 정렬된 배열과 역순 배열일 때 퀵 정렬의 속도가 현저히 느려지는 것은 축값으로서 가장 오른쪽의 값을 택하기 때문

-> 가장 큰값과 작은 값이 축값이 되기 때문

문제의 해결 방법

- 축 값을 잘 선택하는 것이다.

-> 그 대안으로 축값을 난수(random)로 선택하는 것이다.

확률에 퀵 정렬의 속도 향상을 기대해보는 것이다.

|  |
| --- |
| \*\* 난수을 이용한 퀵정렬 알고리즘 |
| void quick\_sort2(int a[], int n)  {  int v, t;  int i, j;  int l, r;  init\_stack();  l = 0;  r = n-1;  push(r);  push(l);  while(!is\_stack\_empty())  { // 난수로 분할값을 결정  l = pop();  r = pop();  if(r-l+1>1)  {  r = random(r-l+1) + l; // 난수 발생  v = a[t];  a[t] = a[r];  a[r] = v;  i = l-1;  j = r;  while(1)  {  while(a[++i] < v);  while(a[--j] > v);  if(i >= j) break;  t = a[i];  a[i] = a[j];  a[j] = t;  }  t = a[i];  a[i] = a[r];  a[r] = t;  push(r);  push(i+1);  push(i-1);  push(l);  }  }  } |

- 엄첨나게 속도가 빨라졌다.

- 고르게 성능이 좋아진다.

5.7.7 퀵 정렬의 개선 3: 삽입 정렬

- 삽입 정렬을 사용하는 것은 퀵 정렬의 재귀의 깊이를 상당히 줄여줌으로써 메모리의 부족에 대한 문제를 어느 정도 해결하게 해준다.

|  |
| --- |
|  |
| void quick\_sort3(int a[], int n)  {  int v, t;  int r, l;  int i, j;  init\_stack();  l = 0;  r = n-1;  push(l);  push(r);  while(!is\_stack\_empty())  {  l = pop();  r = pop();  if(r-l+1 > 200) // 구간의 크기가 200이상일 때만 퀵 정렬  {  t = random(r-l+1) + l;  v = a[t];  a[t] = a[r];  a[r] = v;  i = l-1;  j = r;  while(1)  {  while(a[++i] < v);  while(a[--j] > v);  if(i >= j) break;  t = a[i];  a[i] = a[j];  a[j] = t;  }  t = a[i];  a[i] = a[r];  a[r] = t;  push(r);  push(i+1);  push(i-1);  push(l);  }  else // 200보다 작은 크기에는 삽입 정렬  insert\_sort(a+l, r-l+1);  }  } |

- 10000 크기의 배열까지 정렬하는데 걸리는 시간이 상당이 짧다.

5.7.8 퀵 정렬의 개선 4: 세 값의 중위

- 세 값의 중위(Three of Median)를 이용하여 퀵 정렬을 개선시키는 방법도 있다.

- 세 값은 왼쪽의 값 a[l]과 가장 오른쪽의 값a[r]과 중간 값a[(r+l)/2]

|  |
| --- |
| \*\* |
| void quick\_sort4(int a[], int n)  {  int v, t;  int i, j;  int l, r;  init\_stack();  l = 0;  r = n-1;  push(r);  push(l);  while(!is\_stack\_empty())  {  l = pop();  r = pop();  if(r-l+1 > 200)  {  t = (r+l) >> 1; // t는 중앙  if(a[l] > a[t]) // 이후 왼쪽, 중앙, 오른쪽 값을 정렬  {  v = a[l];  a[l] = a[t];  a[t] = v;  }  if(a[l] > a[r])  {  v = a[l];  a[l] = a[r];  a[r] = v;  }  if(a[t] > a[r])  {  v = a[t];  a[t] = a[r];  a[r] = v;  }  v = a[t];  a[t] = a[r-1];  a[r-1] = v;  i = l;  j = r-1;  while(1)  {  while (a[++i] < v);  while (a[--j] > v);  if (i >= j) break;  t = a[i];  a[i] = a[j];  a[j] = t;  }  t = a[i];  a[i] = a[r-1];  a[r-1] = t;  push(r);  push(i+1);  push(i-1);  push(l);  }  else  insert\_sort(a+1, r-l+1);  }  } |

- 최악의 경우에 가장 빠른 속력을 나타낸다.

5.7.9 qsort 함수

- Turbo C에서는 막강한 퀵 정렬 함수를 라이브러리 함수로 제공한다.

- 매뉴얼에 의하면 세 값의 중위(Three of Medium)법을 이용한다.

5.8 기수 정렬(Radix Sort)

- 기수 정렬은 컴퓨터에서 사용되는 자료가 디지털(digital)자료임에 착안하여 고안된 정렬 방법이다.

- 2진수 자료에 대해 매우 빠른 속도로 정렬할 수 있는 방법을 이 절에서 소계함

- 기수 교환 정렬(Radix Exchange Sort)은 비트 별로 분할하여 재귀호출하는 모습이 마치 퀵 정렬과 비슷하다.

- 키의 비트를 왼쪽에서 오른쪽으로 검사하면서 비트별로 분할을 하기 때문에 최악의 경우가 없다는 것이 장점이다.

- 직접 기수 정렬(Straight Radix Sort)는 분포수세기(Distribution)를 활용한 것으로 기수 교환 정렬과는 반대로 오른쪽에서 왼쪽으로 비트를 검사하면서 분포수세기를 한다.

5.8.1 기수 교환 정렬(Radix Exchange Sort)의 전략

- 기수 교환 정렬은 비트를 다룬다는 것을 빼고는 퀵 정렬과 유사하다.

<기수 교환 정렬 알고리즘(a, N, b)>

1. 만약 n>1이고 b>=0이면

1.1 N 크기의 a배열을 분할하여 비트 1이 시작되는 곳(분할지점)을 j에 할당

1.2 기수 교환 정렬 알고리즘(a, j, b-1)

1.3 기수 교환 정렬 알고리즘(a+j, N-j, b-1)

5.8.2 기수 교환 정렬의 실제

- p.376 그림참조

- 멀리 떨어진 요소를 교환하기 때문에 안전성이 없다.

5.8.3 C로 구현한 기수 교환 정렬

- 우선 비트를 다루는 방법을 알아야 한다.

- C는 비트별로 제어를 하기 위한 다양한 비트 연산자를 제공하므로 이런 함수를 쉽게 만들 수 있다.

|  |
| --- |
| \*\*j개의 비트를 뽑아내는 함수 |
| unsigned bits(unsigned x, int k, int j)  {  return (x >> k) & ~(~0 << j);  } |

|  |
| --- |
| \*\*기수 교환 정렬 |
| void radix\_exchange\_sort(int a[], int n, int b)  {  int t, i, j;  if(n>1 && b>=0)  {  i = 0; // i는 첫 레코드  j = n-1; // j는 마지막 레코드  while(1)  {  while(bits(a[i], b, 1) == 0 && i < j) i++; // 검색  while(bits(a[j], b, 1) != 0 && i < J) i--;  if(i >= j) break; // i와 j가 같아지면 끝이다.  t = a[i]; // i요소와 j요소를 교환한다.  a[i] = a[j];  a[j] = t;  }  if (bits(a[n-1], b, 1) == 0) j++; // 비트가 모두 0인 경우  radix\_exchange\_sort(a, j, b-1); // 다음 비트와 분할에 대해 재귀  radix\_exchange\_sort(a+j, n-j, b-1);  }  } |

5.8.4 기수 교환 정렬의 분석

- 기수 교환 정렬은 퀵 정렬보다는 느리지만 O(NlogN)의 효율적인 정렬 알고리즘이다.

- 퀵 정렬보다 느린 이유는 비트를 검출하기 위해서 복잡한 bits 함수를 호출하기 때문

- 기수 교환 정렬 최악의 경우 재귀의 깊이는 키값의 비트수 만큼 밖에 되지 않는다.

- 정수 배열과 같이 비트수가 16 정도라면 효율이 있지만 만약 키 값이 문자열이라면 검색할 비트수는 문자수x8만큼이 된다.

- 기수 교환 정렬은 키 값이 작은 비트수로 이루어져야 하고 일정해야 하는 사용상의 제한이 있다.

5.8.5 기수 교환 정렬의 개선(비재귀판)

- 비재귀판을 만드는 의의는 이제까지와 마찬가지로 속도의 향상과 프로그램의 안전에 있다.

|  |
| --- |
| \*\* |
| void radix\_exchange\_sort1(int a[], int n)  {  int t, i, j;  int l, r;  int b;  init\_stack();  b = 15;  l = 0;  r = n-1;  push(b);  push(r);  push(l);  while(!is\_stack\_empty())  {  l = pop();  r = pop();  b = pop();  if(r > l && b >= 0)  {  i = l;  j = r;  while(1)  {  while(bits(a[i], b, 1) == 0 && i < j) j++;  while(bits(a[j], b, 1) != 0 && i < j) j--;  t = a[i];  a[i] = a[j];  a[j] = t;  }  if(bits(a[r], b, 1) == 0) j++; // 마지막 비트가 0이면 j증가  push(b-1); // 뒷쪽 분할에 대한 재귀  push(r);  push(j);  push(b-1); // 앞쪽 분할에 대한 재귀  push(j-1);  push(l);  }  }  } |

5.8.6 직접 기수 정렬(Straight Radix Sort)의 전략

- 집접 기수 정렬은 특정한 경우에서 이제까지의 정렬 방법중에서 가장 빠른 속도를 나타낸다.

|  |  |
| --- | --- |
| 기수 교환 정렬 | 왼쪽에서 오른쪽으로 비트를 검사함  분할에 의한 재귀호출로 정렬  약간의 스택만 추가적으로 필요 |
| 직접 기수 정렬 | 오른쪽에서 왼쪽으로 비트를 검사함  분포수세기에 의한 비재귀적 방법으로 정렬  배열과 정렬할 배열과 똑 같은 크기의 배열이 필요 |

- 직접 기수 정렬은 정렬 알고리즘 중에서 가장 많은 추가적인 메모리가 필요하지만 속도는 가장 빠르다.

-> 따라서 메모리와 속도의 타협이 필요하다.

- 분포수세기는 키의 범위가 적고 이산적(discrete)이며 그래서 같은 키가 많이 중복되는 경우에 가장 효율적으로 정렬할 수 있는 방법이다.

- 또한 분포수세기는 안정적인 정렬 알고리즘이다.

- 직접 기수 정렬은 비트의 수가 0과 1로 2개임으로 범위가 작아 중복이 많이 된다.

- 따라서 비트의 나열을 정렬할 때에 분포수세기가 적정하다.

5.8.7 직접 기수 정렬의 실제

If 8비트 -> 배열의 크기 2 -> 분포수세기 실행 8번

If 배열의 크기 4 -> 분포수세기 실행 4번

-> 배열의 크기와 분포수세기의 횟수는 서로 반비례하다.

5.8.8 c로 구현한 직접 기수 정렬

직접 기수 정렬은 c로 구현하기에 앞서 몇가지 상수를 정의하자

W = 키의 전체 비트수

If int형 키, w= 16,

M = 한번에 검색하는 비트수

MV = 2에 n승 즉 count[]배열의 크기를 의미한다.

|  |
| --- |
| \*\* |
| #define W 16  #define M 4  #define MV (1 << M)  void straight\_radix\_sort(int a[], int n)  {  int i, j;  int \*b, \*count;  b = (int\*)malloc(sizeof(int)\*n);  count = (int\*)malloc(sizeof(int)\*MV);  for(i=0; i<W/M; j++) // W/M == 4번 검색한다.  {  for(j=0; j<MV; j++) // count[] 배열의 초기화  count[j]=0;  for(j=0; j<n; j++) // 분포수세기  count[bits(a[j], i\*M, M)]++;  for(j=1; j<MV; j++) // 누적분포구하기  count[j]=count[j]+count[j-1];  for(j=n-1; j>=0; j--) // b배열에 정렬  b[--count[bits(a[j], i\*M, M)]] = a[j];  for(j=0; j<n; j++) // a 배열에 b배열을 복사  a[j]=b[j];  }  free(b);  free(count);  } |

5.8.9 직접 기수 정렬의 분석

- 직접 기수 정렬은 입력 자료에 전혀 무관한다.

- 어떤 자료가 입력되더라도 같은 개수의 자료라면 똑 같은 시간을 나타낸다.

- 또한 속도가 무척 빠르다.

- 일반적으로 W의 1/4정도로 M을 잡는 것을 권장한다.

- 분포수세기가 O(N)의 성능을 가지기 때문에 직접 기수 정렬도 마찬가지로 O(N)의 성능을 가진다.

- if문이 없는 정렬법인 분포수세기는 빠른 속도를 나타내지만 너무 많은 메모기라 필요하여 사용이 제한된다.

5.8.9 두가지 기수 정렬법의 일반화

- 다른 정렬법과는 달리 두 가지 기수 정렬법은 일반화하기가 껄그럽다.

-> 근본부터 제한된 정렬키에 대해서만 사용할 수 있기 때문이다.

5.9 힙 정렬(Heap Sort)

- 힙(Heap)은 우선순위 큐(Priority Queue)의 일종으로 우선순위가 높은 요소를 효율적으로 선택할 수 있는 자료 구조

- 본서는 힙은 나무 구조로 구현하며 이 나무 구조는 배열을 이용하여 구현한다.

- 힙 정렬은 부가적인 메모리(extra memory)가 전혀 필요없으면서도 O(NlogN)의 성늘을 가지는 매우 빠른 정렬법이다.

- 입력 자료에도 거의 무관하게 고른 성능을 보여주는 뛰어난 성능을 가지고 있다.

5.9.1 우선순위 큐(Priority Queue)란

- 자료의 집합이 있으며 이 자료의 각각의 순위(Priority)가 있다.

1. 생성(Construct): 주어진 N개의 자료로 우선순위 큐를 생성한다.

2. 삽입(Insert): 우선순위 큐에 새로운 자료 삽입

3. 제거(Remove): 순위가 가장 높은 자료를 제거

4. 대치(Replace): 순위가 가장 높은 자료와 새로운 자료를 대치

5. 변경(Change): 자료의 순위를 변경한다.

6. 삭제(Delete): 임의의 자료를 삭제한다.

7. 결합(Join): 두 우선순위 큐를 결합하여 큰 우선순위 큐를 만든다.

- 우선순위 큐는 순위에 의해 순서가 유지되며 관리되는 자료구조의 추상적인 개념

5.9.2 힙(Heap)이란?

- p.388

5.9.3 나무 구조를 배열로 구현하는 법

- 나무 구조를 구현하는 방법에는 연결 리스트를 이용하는 방법과 배열을 이용하는 방법이 있다.

- 연결 리스트는 부모와 자식간의 관계를 포인터를 이용하여 직접적으로 결합시킨다.

- 배열을 이용하는 방법은 첨자(index)의 조작으로 간접적으로 부모 자식간을 연결시켜준다.

- 이진 나무 구조의 경우 부모와 자식간의 관계를 나타낼 수 있다.

1. 번호 j를 갖는 노드의 부모의 번호는 j/2이다.

2. 번호 j를 갖는 노드의 왼쪽 자식의 번호는 j\*2이고, 오른쪽 자식은 j\*2+1이다.

3. 힙 자료의 개수 n의 절반(n/2)까지가 내부 노드이다.