데이터 구조 9장 실습과제

20223100 박신조

8.14 이진트리를 이용한 영어 사전 프로그램

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <memory.h>

#define MAX_WORD_SIZE    100
#define MAX_MEANING_SIZE 200

typedef struct {
    char word[MAX_WORD_SIZE];
    char meaning[MAX_MEANING_SIZE];
} element;

typedef struct TreeNode {
    element key;
    TreeNode* left, * right;
} TreeNode;

int compare(element e1, element e2)
{
    return strcmp(e1.word, e2.word);
}

void display(TreeNode* p)
{
    if (p != NULL) {
        printf("(");
        display(p->left);
        printf("%s:%s", p->key.word, p->key.meaning);
        display(p->right);
        printf(")");
    }
}

TreeNode* search(TreeNode* root, element key)
{
    TreeNode* p = root;
    while (p != NULL) {
        if (compare(key, p->key) == 0)
            return p;
        else if (compare(key, p->key) < 0)
            p = p->left;
        else if (compare(key, p->key) > 0)
            p = p->right;
    }
    return p;
}

TreeNode* new_node(element item)
{
    TreeNode* temp = (TreeNode*)malloc(sizeof(TreeNode));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

TreeNode* insert_node(TreeNode* node, element key)
{
    if (node == NULL) return new_node(key);
    if (compare(key, node->key) < 0)
        node->left = insert_node(node->left, key);
    else if (compare(key, node->key) > 0)
        node->right = insert_node(node->right, key);
    return node;
}

TreeNode* min_value_node(TreeNode* node)
{
    TreeNode* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}
```

```c
TreeNode* delete_node(TreeNode* root, element key)
{
    if (root == NULL) return root;
    if (compare(key, root->key) < 0)
        root->left = delete_node(root->left, key);
    if (compare(key, root->key) > 0)
        root->right = delete_node(root->right, key);
    else {
        if (root->left == NULL) {
            TreeNode* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            TreeNode* temp = root->left;
            free(root);
            return temp;
        }
        TreeNode* temp = min_value_node(root->right);
        root->key = temp->key;
        root->right = delete_node(root->right, temp->key);
    }
    return root;
}

void help()
{
    printf("\n**** i: 입력, d: 삭제, s: 탐색, p: 출력, q: 종료 ****: ");
}

int main()
{
    char command;
    element e;
    TreeNode* root = NULL;
    TreeNode* tmp;

    do {
        help();
        command = getchar();
        getchar();
        switch (command) {
        case 'i':
            printf("단어:");
            gets(e.word);
            printf("의미:");
            gets(e.meaning);
            root = insert_node(root, e);
            break;
        case 'd':
            printf("단어:");
            gets(e.word);
            root = delete_node(root, e);
            break;
        case 'p':
            display(root);
            printf("\n");
            break;
        case 's':
            printf("단어:");
            gets(e.word);
            tmp = search(root, e);
            if (tmp != NULL)
                printf("의미:%s\n", e.meaning);
            break;
        }

    } while (command != 'q');
    return 0;
}
```

```
**** i: 입력, d: 삭제, s: 탐색, p: 출력, q: 종료 ****: i
단어:tree
의미:나무

**** i: 입력, d: 삭제, s: 탐색, p: 출력, q: 종료 ****: i
단어:student
의미:학생

**** i: 입력, d: 삭제, s: 탐색, p: 출력, q: 종료 ****: i
단어:information
의미:정보

**** i: 입력, d: 삭제, s: 탐색, p: 출력, q: 종료 ****: s
단어:student
의미:정보
```

이진트리를 활용하여 영어사전을 구현한 코드

#define MAX_MEANING_SIZE 200

MAX_MEANING_SIZE를 200으로 지정

typedef struct { . . . } element;

단어와 뜻을 저장하는 구조체
멤버 - word : 단어 , meaning : 의미

typedef struct TreeNode { . . . } TreeNode;

트리의 노드 역할을 하는 구조체
멤버 - key : 저장할 데이터 , left, right : 자식노드

int compare(element e1, element e2)

두 단어를 비교하는 함수
strcmp 함수를 사용해 e1.word와 e2.word를 비교

void display(TreeNode* p)

트리의 모든 노드를 중위 순회(inorder traversal) 방식으로 출력하는 함수
왼쪽 자식 → 현재 노드 → 오른쪽 자식 순서로 출력
출력 형식: (왼쪽서브트리)(노드)(오른쪽서브트리)

TreeNode* search(TreeNode* root, element key)

트리에서 특정 단어(key.word)를 탐색하는 함수
루트부터 시작하여 현재 노드, 왼쪽/오른쪽 자식 순으로 이동하며 탐색
일치하는 단어를 찾으면 해당 노드 포인터 반환, 없으면 NULL 반환

TreeNode* new_node(element item)

새로운 트리 노드를 생성하는 함수
동적으로 메모리 할당하고, key 필드에 item 저장
left와 right 포인터는 NULL로 초기화
새로 생성한 노드 리턴

TreeNode* insert_node(TreeNode* node, element key)

이진 탐색 트리에 새로운 노드를 삽입하는 함수
재귀적으로 비교하며 적절한 위치를 찾아 삽입
이미 존재하는 단어는 삽입하지 않고 현재 트리 상태 유지

TreeNode* min_value_node(TreeNode* node)

트리에서 가장 작은 값을 가진 노드(왼쪽 끝 노드)를 반환하는 함수

TreeNode* delete_node(TreeNode* root, element key)

트리에서 재귀적으로 특정 단어를 찾아 삭제하는 함수
자식이 없음 → 바로 삭제
자식이 하나 → 그 자식 노드를 대체
자식이 둘 → 오른쪽 서브트리에서 가장 작은 노드를 찾아 대체

void help()

사용자 명령 안내 메시지를 출력하는 함수

int main()

do while 문을 통해 사용자의 입력대로 해당 함수를 호출하여 동작 수행
삽입, 삭제, 탐색, 트리구조 출력 등이 가능함.
getchar()를 통해 한 문자만 입력받아
switch문으로 메뉴별로 구분해줌

사용자 입력 메뉴
'i': 단어와 의미를 입력받아 트리에 삽입
'd': 단어를 입력받아 트리에서 삭제
's': 단어를 입력받아 트리에서 검색 후 의미 출력
'p': 전체 트리 출력
'q': 프로그램 종료

9.3 히프트리 삭제, 삽입 함수

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_ELEMENT 200

typedef struct {
    int key;
} element;
typedef struct {
    element heap[MAX_ELEMENT];
    int heap_size;
} HeapType;

HeapType* create()
{
    return (HeapType*)malloc(sizeof(HeapType));
}

void init(HeapType* h)
{
    h->heap_size = 0;
}

void insert_max_heap(HeapType* h, element item)
{
    int i;
    i = ++(h->heap_size);

    while ((i != 1) && (item.key > h->heap[i / 2].key)) {
        h->heap[i] = h->heap[i / 2];
        i /= 2;
    }
    h->heap[i] = item;
}
element delete_max_heap(HeapType* h)
{
    int parent, child;
    element item, temp;

    item = h->heap[1];
    temp = h->heap[(h->heap_size)--];
    parent = 1;
    child = 2;
    while (child <= h->heap_size) {
        if ((child < h->heap_size) &&
            (h->heap[child].key) < h->heap[child + 1].key)
            child++;
        if (temp.key >= h->heap[child].key)
            break;
        h->heap[parent] = h->heap[child];
        parent = child;
        child += 2;
    }
    h->heap[parent] = temp;
    return item;
}

int main()
{
    element e1 = { 10 }, e2 = { 5 }, e3 = { 30 };
    element e4, e5, e6;
    HeapType* heap;

    heap = create();
    init(heap);

    insert_max_heap(heap, e1);
    insert_max_heap(heap, e2);
    insert_max_heap(heap, e3);

    e4 = delete_max_heap(heap);
    printf("< %d > ", e4.key);
    e5 = delete_max_heap(heap);
    printf("< %d > ", e5.key);
    e6 = delete_max_heap(heap);
    printf("< %d > \n", e6.key);

    free(heap);
    return 0;
}
```

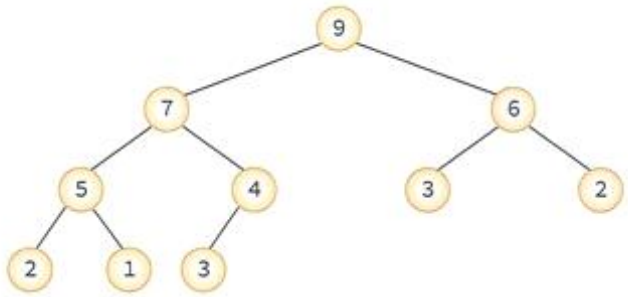
```
< 30 > < 10 > < 5 >
```

최대 히프트리를 배열을 통해 구현하고 삽입, 삭제 함수를 적용한 코드

최대 히프란

여러 개의 값들 중 가장 큰 값이나 가장 작은 값을 빠르게 찾아내도록 만들어진 자료구조이
즉 부모 노드의 값이 자식 노드의 값보다 크거나 같은 완전이진트리

이와 반대되는 개념으로 최소히프가 있다.
- 부모의 노드값이 자식의 노드값보다 항상작은



최대 히프

#define MAX_ELEMENT 200

MAX_ELEMENT을 200으로 지정

typedef struct { . . . } element;

히프트리의 데이터를 저장하는 구조체
멤버 - key : 정수를 저장

typedef struct { . . . } HeapType;

히프트리의 노드 역할을 하는 구조체
멤버 - heap : 완전이진트리를 나타낼 배열
heap_size : 히프의 크기를 나타내는 변수

HeapType* create()

동적으로 메모리를 할당하여 새로운 히프를 생성하는 함수
HeapType 구조체 크기만큼 할당한 메모리를 포인터로 지정하여 리턴

void init(HeapType* h)

히프를 초기화하는 함수
heap_size를 0으로 설정하여 공백상태로 설정

void insert_max_heap(HeapType* h, element item)

히프에 새로운 요소를 삽입하는 함수
heap_size를 1 증가
새로 추가될 위치부터 부모 노드와 비교하여
item.key > h->heap[i / 2].key)일때까지 i/2로 하여 위쪽 인덱스로 이동
조건을 만족하는 위치를 찾아 item을 삽입

element delete_max_heap(HeapType* h)

히프의 루트(최댓 값)를 삭제하고 재정렬하는 함수
루트 노드 값을 item에 저장(리턴할 값)
마지막 노드를 꺼내 temp에 저장하고 히프 크기를 줄임
루트부터 자식 노드와 비교하여 더 큰 자식 쪽으로 내려가며 재정렬
적절한 위치에 temp 삽입
이후 삭제된 item 반환

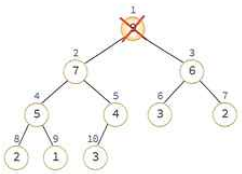
자식의 인덱스를 알고 싶을 때
- **왼쪽 자식의 인덱스 = (부모의 인덱스) * 2**
- **오른쪽 자식의 인덱스 = (부모의 인덱스) * 2+ 1**

부모의 인덱스를 알고 싶을 때
- **부모의 인덱스 = (자식의 인덱스)/2**

int main()

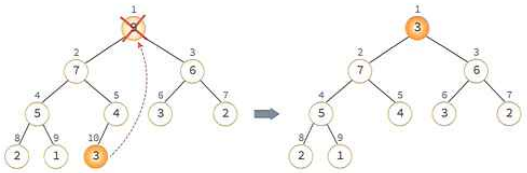
craete() 함수를 통해 히프를 선언해주고
insert_max_heap() 함수를 통해 요소를 추가
delete_max_heap() 함수를 통해 요소를 삭제 후 출력
모든 동작이 끝난 후엔 free를 통해 할당받은 메모리 반납

삭제연산



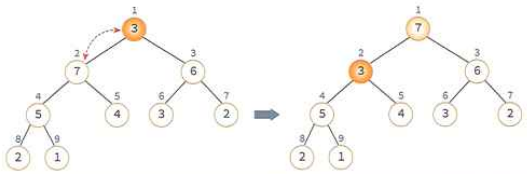
[그림 9-14] 히프트리의 삭제연산

(1) 먼저 루트 노드가 삭제된다. 빈 루트 노드 자리에는 히프의 마지막 노드를 가져온다.



[그림 9-15] 히프트리의 삭제연산 #1

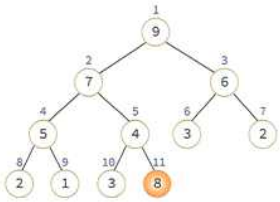
(2) 새로운 루트인 3과 자식 노드들을 비교해보면 자식 노드가 더 크기 때문에 교환이 일어난다. 자식 중에서 더 큰 값과 교환이 일어난다. 따라서 3과 7이 교환된다.



[그림 9-16] 히프트리의 삭제연산 #2

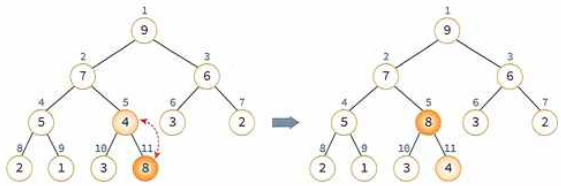
삽입연산

(1) 먼저 번호순으로 가장 마지막 위치에 이어서 새로운 요소 8이 삽입된다.



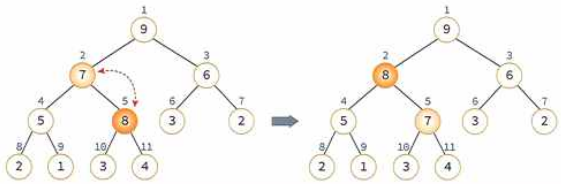
[그림 9-10] 히프트리의 삽입연산 단계 #1

(2) 부모 노드 4와 비교하여 삽입 노드 8이 더 크므로 교환한다.



[그림 9-11] 히프트리의 삽입연산 단계 #2

(3) 부모 노드 7과 비교하여 삽입 노드 8이 더 크므로 교환한다.



[그림 9-12] 히프트리의 삽입연산 단계 #3

(4) 삽입 노드 8이 부모 노드 9보다 작으므로 더 이상 교환하지 않는다.

9.4 히프정렬 프로그램

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_ELEMENT 200
#define SIZE 8

typedef struct {
    int key;
} element;

typedef struct {
    element heap[MAX_ELEMENT];
    int heap_size;
} HeapType;

HeapType* create( )
{
    return (HeapType*)malloc(sizeof(HeapType));
}

void init(HeapType* h)
{
    h->heap_size = 0;
}

void insert_max_heap(HeapType* h, element item)
{
    int i;
    i = ++(h->heap_size);

    while ((i != 1) && (item.key > h->heap[i / 2].key)) {
        h->heap[i] = h->heap[i / 2];
        i /= 2;
    }
    h->heap[i] = item;
}
element delete_max_heap(HeapType* h)
{
    int parent, child;
    element item, temp;

    item = h->heap[1];
    temp = h->heap[(h->heap_size)--];
    parent = 1;
    child = 2;
    while (child <= h->heap_size) {
        if ((child < h->heap_size) &&
            (h->heap[child].key) < h->heap[child + 1].key)
            child++;
        if (temp.key >= h->heap[child].key)
            break;
        h->heap[parent] = h->heap[child];
        parent = child;
        child *= 2;
    }
    h->heap[parent] = temp;
    return item;
}

void heap_sort(element a[], int n)
{
    int i;
    HeapType* h;

    h = create( );
    init(h);
    for (i = 0; i < n; i++) {
        insert_max_heap(h, a[i]);
    }
    for (i = (n - 1); i >= 0; i--) {
        a[i] = delete_max_heap(h);
    }
    free(h);
}

int main( )
{
    element list[SIZE] = { 23, 56, 11, 9, 56, 99, 27, 34 };
    heap_sort(list, SIZE);
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", list[i].key);
    }
    printf("\n");
    return 0;
}
```

```
9 11 23 27 34 56 56 99
```

히프를 사용하여 오름차순으로 정렬하는 코드(히프정렬)

#define MAX_ELEMENT 200

MAX_ELEMENT을 200으로 지정

#define SIZE 8

SIZE를 8로 지정

typedef struct { . . . } element;

히프트리의 데이터를 저장하는 구조체
멤버 - key : 정수를 저장

typedef struct { . . . } HeapType;

히프트리의 노드 역할을 하는 구조체
멤버 - heap : 완전이진트리를 나타낼 배열, heap_size : 히프의 크기를 나타내는 변수

HeapType* create()

동적으로 메모리를 할당하여 새로운 히프를 생성하는 함수
HeapType 구조체 크기만큼 할당한 메모리를 포인터로 지정하여 리턴

void init(HeapType* h)

히프를 초기화하는 함수
heap_size를 0으로 설정하여 공백상태로 설정

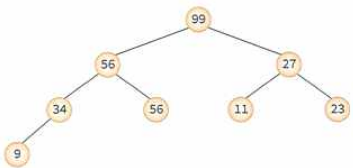
void insert_max_heap(HeapType* h, element item)

히프에 새로운 요소를 삽입하는 함수
heap_size를 1 증가
새로 추가될 위치부터 부모 노드와 비교하여
item.key > h->heap[i / 2].key)일때까지 i/2로 하여 위쪽 인덱스로 이동
조건을 만족하는 위치를 찾아 item을 삽입

element delete_max_heap(HeapType* h)

히프의 루트(최댓 값)를 삭제하고 재정렬하는 함수
루트 노드 값을 item에 저장(리턴할 값)
마지막 노드를 꺼내 temp에 저장하고 히프 크기를 줄임
루트부터 자식 노드와 비교하여 더 큰 자식 쪽으로 내려가며 재정렬
적절한 위치에 temp 삽입
이후 삭제된 item 반환

이 데이터들을 차례대로 최대 히프에 추가하여 다음과 같은 히프를 생성한다.



한 번에 하나씩 요소를 히프에서 꺼내서 배열의 뒤쪽부터 저장하면 된다. 배열 요소들은 값이 증가되는 순서로 정렬되게 된다.

void heap_sort(element a[], int n)

배열 a[]에 있는 요소들을 히프 정렬하여 오름차순으로 정렬하는 함수
최대 히프 할당 및 초기화
배열의 모든 요소를 히프에 삽입
히프에서 가장 큰 원소부터 하나씩 꺼내서 배열 끝부터 채움
정렬이 모두 끝나면 히프 반환

int main()

정렬되지 않은 배열 설정
heap_sort() 함수를 호출하여 오름차순으로 배열 정렬
이후 정렬된 배열 출력

히프 정렬의 시간복잡도

히프트리의 전체 높이가는 거의 log2이다. - 완전이진트리이기 때문에
따라서 하나의 요소를 히프에 삽입하거나 삭제할 때 히프를 재정비하는 시간이 log2n만큼 소요된다.
요소의 개수가 n개이므로 시간복잡도는 O(nlog2n)이다.

삽입연산 , 삭제연산의 시간복잡도는 O(log2n)이다.

9.5 LPT 프로그램

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_ELEMENT 200
#define JOBS 7
#define MACHINES 3

typedef struct {
    int id;
    int avail;
} element;

typedef struct {
    element heap[MAX_ELEMENT];
    int heap_size;
} HeapType;

HeapType* create()
{
    return (HeapType*)malloc(sizeof(HeapType));
}

void init(HeapType* h)
{
    h->heap_size = 0;
}

void insert_min_heap(HeapType* h, element item)
{
    int i;
    i = ++(h->heap_size);

    while ((i != 1) && (item.avail < h->heap[i / 2].avail)) {
        h->heap[i] = h->heap[i / 2];
        i /= 2;
    }
    h->heap[i] = item;
}

element delete_min_heap(HeapType* h)
{
    int parent, child;
    element item, temp;
    item = h->heap[1];
    temp = h->heap[(h->heap_size)--];
    parent = 1;
    child = 2;
    while (child <= h->heap_size) {
        if ((child < h->heap_size) &&
            (h->heap[child].avail) > h->heap[child + 1].avail)
            child++;
        if (temp.avail < h->heap[child].avail)
            break;
        h->heap[parent] = h->heap[child];
        parent = child;
        child *= 2;
    }
    h->heap[parent] = temp;
    return item;
}

int main()
{
    int jobs[JOBS] = { 8, 7, 6, 5, 3, 2, 1 };
    element m = { 0, 0 };
    HeapType* h;
    h = create();
    init(h);

    for (int i = 0; i < MACHINES; i++) {
        m.id = i + 1;
        m.avail = 0;
        insert_min_heap(h, m);
    }

    for (int i = 0; i < JOBS; i++) {
        m = delete_min_heap(h);
        printf("JOB %d을 시간=%d부터 시간=%d까지 기계 %d번에 할당한다. \n",
            i, m.avail, m.avail + jobs[i] - 1, m.id);
        m.avail += jobs[i];
        insert_min_heap(h, m);
    }
    return 0;
}
```

```
JOB 0을 시간=0부터 시간=7까지 기계 1번에 할당한다.
JOB 1을 시간=0부터 시간=6까지 기계 2번에 할당한다.
JOB 2을 시간=0부터 시간=5까지 기계 3번에 할당한다.
JOB 3을 시간=6부터 시간=10까지 기계 3번에 할당한다.
JOB 4을 시간=7부터 시간=9까지 기계 2번에 할당한다.
JOB 5을 시간=8부터 시간=9까지 기계 1번에 할당한다.
JOB 6을 시간=10부터 시간=10까지 기계 2번에 할당한다.
```

최소 히프를 사용하여 LPT알고리즘을 구현한 코드

typedef struct { . . . } element;

저장될 데이터를 나타내는 구조체
멤버 - id : 번호, avail : 사용가능 시간

typedef struct { . . . } HeapType;

히프트리의 노드 역할을 하는 구조체
멤버 - heap : 완전이진트리를 나타낼 배열, heap_size : 히프의 크기를 나타내는 변수

HeapType* create()

동적으로 메모리를 할당하여 새로운 히프를 생성하는 함수
HeapType 구조체 크기만큼 할당한 메모리를 포인터로 지정하여 리턴

void init(HeapType* h)

히프를 초기화하는 함수
heap_size를 0으로 설정하여 공백상태로 설정

void insert_min_heap(HeapType* h, element item)

히프에 새로운 요소를 삽입하는 함수
heap_size를 1 증가
새로 추가될 위치부터 부모 노드와 비교하여
item.key < h->heap[i / 2].key)일때까지 i/2로 하여 위쪽 인덱스로 이동
조건을 만족하는 위치를 찾아 item을 삽입

element delete_min_heap(HeapType* h)

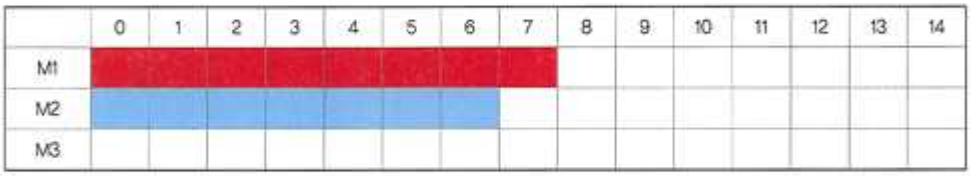
히프의 루트(최솟 값)를 삭제하고 재정렬하는 함수
루트 노드 값을 item에 저장(리턴할 값)
마지막 노드를 꺼내 temp에 저장하고 heap_size를 줄임
루트부터 자식 노드와 비교하여 avail가 더 작은 자식 쪽으로 내려가며 재정렬
이때 자식 노드보다 크면 자식노드를 부모 노드 위치로 올림
적절한 위치에 temp 삽입
이후 삭제된 item 반환

int main()

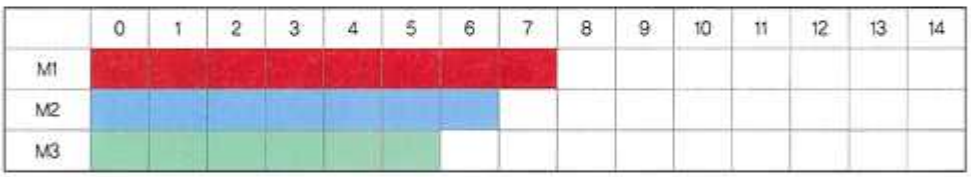
jobs 배열의 요소를 설정
avail가 0인 기계 3대를 히프에 삽입
이후 avail이 작은 기계부터 차례대로 히프에서 받아와
다음 작업이 어느기계에서 언제부터 언제까지 실행되는지 출력
이후 avail를 수정해주고 다시 히프에 삽입

3대의 기계가 실행되는 과정

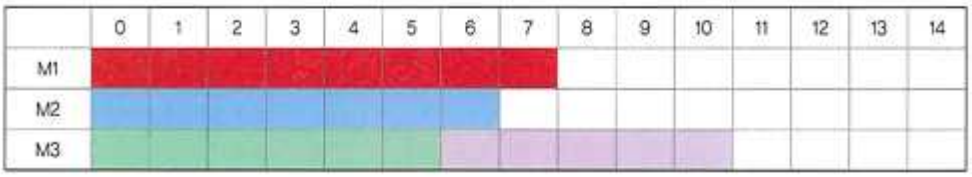
다음 작업은 J2로서 7시간을 차지한다. M2와 M3가 비어 있으므로 M2에 할당된다.



다음 작업은 J3로서 6시간을 차지한다. M3가 비어 있으므로 M3에 할당된다.

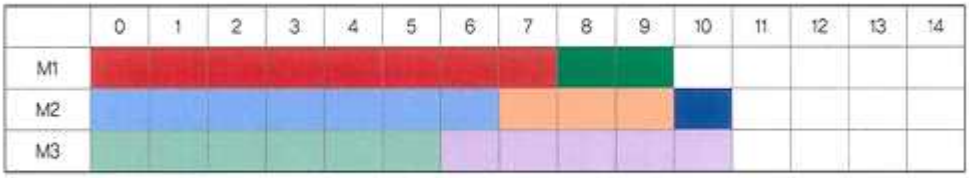


다음 작업은 J4로서 5시간을 차지한다. 가장 먼저 사용가능하게 되는 기계는 M3이므로 M3에 할당된다.



나머지 작업들도 유사한 알고리즘으로 할당된다.

최종적으로 할당된 작업들



9.6 허프만 코드 프로그램(최소 히프 사용)

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_ELEMENT 200

typedef struct TreeNode {
    int weight;
    char ch;
    struct TreeNode* left;
    struct TreeNode* right;
} TreeNode;

typedef struct {
    TreeNode* ptree;
    char ch;
    int key;
} element;

typedef struct {
    element heap[MAX_ELEMENT];
    int heap_size;
} HeapType;

HeapType* create()
{
    return (HeapType*)malloc(sizeof(HeapType));
}

void init(HeapType* h)
{
    h->heap_size = 0;
}

void insert_min_heap(HeapType* h, element item)
{
    int i;
    i = ++(h->heap_size);

    while ((i != 1) && (item.key < h->heap[i / 2].key)) {
        h->heap[i] = h->heap[i / 2];
        i /= 2;
    }
    h->heap[i] = item;
}
element delete_min_heap(HeapType* h)
{
    int parent, child;
    element item, temp;

    item = h->heap[1];
    temp = h->heap[(h->heap_size)--];
    parent = 1;
    child = 2;
    while (child <= h->heap_size) {
        if ((child < h->heap_size) &&
            (h->heap[child].key) > h->heap[child + 1].key)
            child++;
        if (temp.key < h->heap[child].key)
            break;
        h->heap[parent] = h->heap[child];
        parent = child;
        child += 2;
    }
    h->heap[parent] = temp;
    return item;
}

TreeNode* make_tree(TreeNode* left, TreeNode* right)
{
    TreeNode* node =
        (TreeNode*)malloc(sizeof(TreeNode));
    node->left = left;
    node->right = right;
    return node;
}

void destroy_tree(TreeNode* root)
{
    if (root == NULL) return;
    destroy_tree(root->left);
    destroy_tree(root->right);
    free(root);
}

int is_leaf(TreeNode* root)
{
    return !(root->left) && !(root->right);
}
```

```c
void print_tree(TreeNode* root)
{
    if (root != NULL) {
        printf("[%d] ", root->weight);
        print_tree(root->left);
        print_tree(root->right);
    }
}


void print_codes(TreeNode* root, int codes[], int top)
{
    if (root->left) {
        codes[top] = 1;
        print_codes(root->left, codes, top + 1);
    }
    if (root->right) {
        codes[top] = 0;
        print_codes(root->right, codes, top + 1);
    }
    if (is_leaf(root)) {
        printf("%c: ", root->ch);
        print_array(codes, top);
    }
}

void huffman_tree(int freq[], char ch_list[], int n)
{
    int i;
    TreeNode* node, * x;
    HeapType* heap;
    element e, e1, e2;
    int codes[100];
    int top = 0;

    heap = create();
    init(heap);
    for (i = 0; i < n; i++) {
        node = make_tree(NULL, NULL);
        e.ch = node->ch = ch_list[i];
        e.key = node->weight = freq[i];
        e.ptree = node;
        insert_min_heap(heap, e);
    }
    for (i = 1; i < n; i++) {
        e1 = delete_min_heap(heap);
        e2 = delete_min_heap(heap);
        x = make_tree(e1.ptree, e2.ptree);
        e.key = x->weight = e1.key + e2.key;
        e.ptree = x;
        printf("%d+%d->%d \n", e1.key, e2.key, e.key);
        insert_min_heap(heap, e);
    }
    e = delete_min_heap(heap);
    print_codes(e.ptree, codes, top);
    destroy_tree(e.ptree);
    free(heap);
}

int main()
{
    char ch_list[] = { 's', 'i', 'n', 't', 'e' };
    int freq[] = { 4, 6, 8, 12, 15 };
    huffman_tree(freq, ch_list, 5);
    return 0;
}
```

```
4+6->10
8+10->18
12+15->27
18+27->45
n: 11
s: 101
i: 100
t: 01
e: 00
```

최소 히프를 사용하여 허프만 코드를 구현한 코드

허프만 코드란?

어떤 문자열에서 사용된 글자의 수를 이진트리로 압축하여 표현하면 허프만 코딩 트리라고 불린다.

빈도수 분석

A	80
B	16
C	32
D	36
E	123
F	22
G	16
H	51
I	71
...	
Z	1

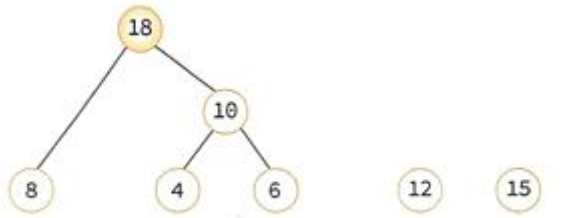
해당 글자가 나오는 수를 빈도수라 하고 이를 이용하여 데이터를 압축하여 표현한다.
압축할 때 보통 우리들이 사용하는 아스키코드를 사용하지 않고 가변길이 코드를 사용한다.
-> 아스키코드로 나타내면 더 많은 메모리가 필요하다.
즉 많이 사용되는(빈도수가 높은) 글자는 짧은 길이의 비트열을 사용하고 빈도수가 낮은 글자는 긴 비트열을 사용한다.

글자	빈도수
e	15
t	12
n	8
i	6
s	4

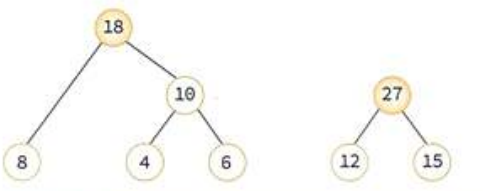
글자	코드	코드길이	빈도수	비트수
e	00	2	15	2*15=30
t	01	2	12	2*12=24
n	11	2	8	2*8=16
i	100	3	6	3*6=18
s	101	3	4	3*4=12
합계				88

이때 teen을 가변코드로 변환하면 01000010이 된다. 그렇지만 가변코드를 활용하였기 때문에
해독하는데 어려움이 있다. -> 3비트씩(일정한 간격x) 끊어서 변경x, 서로 다른 코드를 사용하여 만들었기 때문
그렇지만 코드를 관찰해보면 모든 코드가 다른 코드의 첫 부분이 아니란 것을 알아볼 수 있다.
코딩된 비트열을 왼쪽에서 오른쪽으로 조사하여 보면 정확히 하나의 코드만 일치하는 것을 알 수 있다.
이러한 특수한 코드를 만들기 위하여 이진 트리를 사용하는 것을 호프만 코드(Huffman codes)라고 한다.

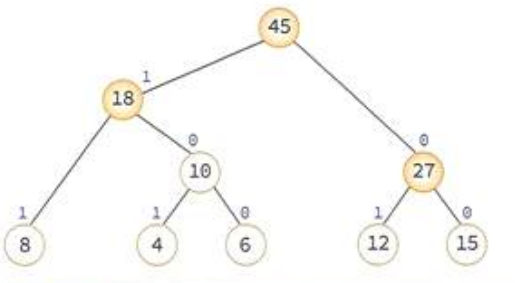
[그림 9-19] 허프만 코드 생성 과정 #1



[그림 9-20] 허프만 코드 생성 과정 #2



[그림 9-21] 허프만 코드 생성 과정 #3



[그림 9-22] 허프만 코드 생성 과정 #4

typedef struct TreeNode { . . . } TreeNode;

이진트리의 노드 역할을 하는 구조체
멤버 - weight : 빈도수, ch : 문자열, left,right : 자식노드를 가르키는 포인터

typedef struct { . . . } element;

저장할 데이터를 나타내는 구조체
멤버 - ptree : 실제 트리 노드를 가르키는 포인터, ch,key는 문자, 빈도수

typedef struct { . . . } HeapType;

최소 히프를 구현한 구조체
멤버 - heap : 히프를 구현한 배열, heap_size : 저장된 데이터 수

HeapType* create()

동적으로 메모리를 할당하여 새로운 히프를 생성하는 함수
HeapType 구조체 크기만큼 할당한 메모리를 포인터로 지정하여 리턴

void init(HeapType* h)

히프를 초기화하는 함수
heap_size를 0으로 설정하여 공백상태로 설정

void insert_min_heap(HeapType* h, element item)

히프에 새로운 요소를 삽입하는 함수
heap_size를 1 증가
새로 추가될 위치부터 부모 노드와 비교하여
item.key < h->heap[i / 2].key)일때까지 i/2로 하여 위쪽 인덱스로 이동
조건을 만족하는 위치를 찾아 item을 삽입

element delete_min_heap(HeapType* h)

히프의 루트(최솟 값)를 삭제하고 재정렬하는 함수
루트 노드 값을 item에 저장(리턴할 값)
마지막 노드를 꺼내 temp에 저장하고 heap_size를 줄임
루트부터 자식 노드와 비교하여 더 작은 자식 쪽으로 내려가며 재정렬
이때 자식 노드보다 크면 자식노드를 부모 노드 위치로 올림
적절한 위치에 temp 삽입
이후 삭제된 item 반환

TreeNode* make_tree(TreeNode* left, TreeNode* right)

왼쪽과 오른쪽 자식을 받아 새로운 트리 노드 생성 하는 함수
문자와 weight는 별도로 설정하기 때문에 노드 구조만 지정

void destroy_tree(TreeNode* root)

할당받아 만들어진 노드를 모두 반환하는 함수
트리를 후위 순회하며 재귀적으로 모든 노드를 반환함

int is_leaf(TreeNode* root)

단말노드인지 판별하는 함수
단말노드라면 1 반환 아니면 0 반환

void print_array(int codes[], int n)

배열의 요소를 출력하는 함수
codes의 0번째 배열부터 n-1인덱스 까지 출력

void print_tree(TreeNode* root)

트리의 현재 상태를 출력하는 함수
중위순회를 활용하여 트리 출력

void print_codes(TreeNode* root, int codes[], int top)

트리를 배열에 저장하는 함수
트리의 왼쪽(1) 오른쪽(0) 경로를 따라가며 codes배열에 저장
만약 단말노드라면 ch와 codes배열 출력

void huffman_tree(int freq[], char ch_list[], int n)

주어진 문자와 빈도수로 허프만 트리 생성하고 출력하는 함수
문자와 빈도수로 초기 트리 노드 생성 -> 히프에 삽입
히프에서 최소값 2개 꺼내 새 트리 노드로 병합
다시 히프에 삽입
최종적으로 루트 노드가 완성됨
모든 과정이 끝난 후
만들어진 트리를 배열에 저장 및 할당받은 트리의 노드와 heap 포인터 메모리 반환

int main()

문자와 빈도수를 설정해준 후 huffman_tree()함수를 호출하여 허프만 코드를 생성

교수님이 오류 있다고 하신 부분
element delete_min_heap(HeapType* h)에서 while 문 속 if문에서 최소 히프 삭제인데 최대 히프 삭제에서 사용한 (child < h->heap_size)로 조건을 설정하여 오류 발생 -> 수정하고 결과를 내면 아래와 같이 출력됨

```
element delete_min_heap(HeapType* h)
{
    int parent, child;
    element item, temp;

    item = h->heap[1];
    temp = h->heap[(h->heap_size)--];
    parent = 1;
    child = 2;
    while (child <= h->heap_size) {
        if ((child > h->heap_size) &&
            (h->heap[child].key) > h->heap[child + 1].key)
            child++;
        if (temp.key < h->heap[child].key)
            break;
        h->heap[parent] = h->heap[child];
        parent = child;
        child += 2;
    }
    h->heap[parent] = temp;
    return item;
}
```

```
4+6->10
10+12->22
8+15->23
22+23->45
s: 111
i: 110
t: 10
n: 01
e: 00
```