

자 료 구 조

- Homework #3 -

(Heap, Heap', Heap'')

* Visual Studio 2015 environment



담당 교수님 : 김용혁 교수님

제 출 날 짜 : 2017. 11. 28.

학 과 : 컴퓨터소프트웨어학과

학 번 : 2014707040

이 름 : 유 진 혁

목 차

1. HeapNode 클래스

- (1). private 멤버 변수
- (2). 생성자
- (3). getData(), setData() 함수
- (4). operator <, operator > 오버로딩

2. HeapArray 클래스

- (1). private 멤버 변수
- (2). 생성자
- (3). getParent(), getLeft(), getRight() 함수
- (4). empty(), full() 함수
- (5). add() 함수
- (6). remove() 함수
- (7). print1() 함수
- (8). not_rotated(), print2() 함수
- (9). H(), H_center(), hor_depth(), vert_depth(), print3() 함수

3. main 함수

4. 실행 결과

5. 고찰

1. HeapNode 클래스

```

/* 힙의 각 노드 클래스 */
class HeapNode
{
public:
    HeapNode(char init_data = '0') : data(init_data) {}
    void setData(char new_data) { data = new_data; }
    char getData() { return data; }

    /* 아스키코드 순서와 다르므로 비교연산자 오버로딩 */
    bool operator<(const HeapNode& node)
    {
        if ((data >= 'A' && data <= 'Z') && (node.data >= 'a' && node.data <= 'z'))
            return false;
        else if ((data >= 'a' && data <= 'z') && (node.data >= 'A' && node.data <= 'Z'))
            return true;
        else
            return data < node.data;
    }
    bool operator>(const HeapNode& node)
    {
        if ((data >= 'A' && data <= 'Z') && (node.data >= 'a' && node.data <= 'z'))
            return true;
        else if ((data >= 'a' && data <= 'z') && (node.data >= 'A' && node.data <= 'Z'))
            return false;
        else
            return data > node.data;
    }

private:
    char data;
};

```

<그림 1. HeapNode 클래스 선언>

(1). private 멤버 변수

HeapNode 클래스는 멤버 변수 char형 data 변수를 가지고 있다. 노드가 가지고 있는 data 값을 저장하는 변수이다.

(2). 생성자

```
HeapNode(char init_data = '0') : data(init_data) {}
```

<그림 2. HeapNode 클래스의 생성자>

클래스 객체 생성 시 default로 data 변수에 0이 들어가고 인자를 주어 해당 인자가 data 변수가 되도록 할 수도 있다.

(3). getData(), setData() 함수

```
void setData(char new_data) { data = new_data; }
char getData() { return data; }
```

<그림 3. setData()와 getData() 함수>

setData()는 인자로 받은 char형 값을 private 변수 data에 저장하는 setter 함수이고, getData()는 private 변수 data를 반환하는 getter 함수이다. 둘 다 inline 함수로 정의하였다.

1. HeapNode 클래스

(4). operator <, operator > 오버로딩

```
/* 아스키코드 순서와 다르므로 비교연산자 오버로딩 */
bool operator<(const HeapNode& node)
{
    if ((data >= 'A' && data <= 'Z') && (node.data >= 'a' && node.data <= 'z'))
        return false;
    else if ((data >= 'a' && data <= 'z') && (node.data >= 'A' && node.data <= 'Z'))
        return true;
    else
        return data < node.data;
}

bool operator>(const HeapNode& node)
{
    if ((data >= 'A' && data <= 'Z') && (node.data >= 'a' && node.data <= 'z'))
        return true;
    else if ((data >= 'a' && data <= 'z') && (node.data >= 'A' && node.data <= 'Z'))
        return false;
    else
        return data > node.data;
}
```

DEC	HEX	OCT	Char	DEC	HEX	OCT	Char
43	2B	053	*	86	56	126	V
44	2C	054	.	87	57	127	W
45	2D	055	-	88	58	130	X
46	2E	056	,	89	59	131	Y
47	2F	057	/	90	5A	132	Z
48	30	060	0	91	5B	133	[
49	31	061	1	92	5C	134	\
50	32	062	2	93	5D	135]
51	33	063	3	94	5E	136	^
52	34	064	4	95	5F	137	_
53	35	065	5	96	60	140	`
54	36	066	6	97	61	141	a
55	37	067	7	98	62	142	b
56	38	070	8	99	63	143	c
57	39	071	9	100	64	144	d
58	3A	072	:	101	65	145	e
59	3B	073	;	102	66	146	f
60	3C	074	<	103	67	147	g
61	3D	075	=	104	68	150	h
62	3E	076	>	105	69	151	i
63	3F	077	?	106	6A	152	j
64	40	100	@	107	6B	153	k
65	41	101	A	108	6C	154	l
66	42	102	B	109	6D	155	m
67	43	103	C	110	6E	156	n
68	44	104	D	111	6F	157	o
69	45	105	E	112	70	160	p
70	46	106	F	113	71	161	q
71	47	107	G	114	72	162	r
72	48	110	H	115	73	163	s
73	49	111	I	116	74	164	t
74	4A	112	J	117	75	165	u
75	4B	113	K	118	76	166	v
76	4C	114	L	119	77	167	w
77	4D	115	M	120	78	170	x
78	4E	116	N	121	79	171	y
79	4F	117	O	122	7A	172	z
80	50	120	P	123	7B	173	{
81	51	121	Q	124	7C	174	
82	52	122	R	125	7D	175	}
83	53	123	S	126	7E	176	~
84	54	124	T	127	7F	177	DEL
85	55	125	U				

<그림 4. 비교 연산자 오버로딩>

<그림 5. 아스키 코드표>

과제에서 제시한 문자의 크기 비교 순서는 '숫자 < 영어 소문자 < 영어 대문자'인 반면, 실제 아스키 코드 상으로는 '숫자 < 영어 대문자 < 영어 소문자' 순으로 값이 증가한다. 그렇기 때문에 비교 연산자를 오버로딩하여 영어 대문자와 소문자의 비교 결과를 뒤바꿔주었다. 클래스의 멤버로 오버로딩하여 인자로 HeapNode 객체를 하나 받고 객체끼리 비교 연산을 했을 때, 안의 data 값을 확인하여 정의된 결과가 반환되도록 하였다. data의 값이 영어 대문자나 소문자가 아닌 경우에는 기존 아스키 코드의 크기순으로 비교하여 그 결과가 반환된다.

2. HeapArray 클래스

```

/* 노드의 배열로 정의된 힙 클래스 */
class HeapArray
{
public:
    HeapArray() : size(0) {}
    void add(char data);
    void remove();
    HeapNode& getParent(int i) { return node[i / 2]; } // i번째 요소의 부모 노드 반환
    HeapNode& getLeft(int i) { return node[i * 2]; } // i번째 요소의 왼쪽 자식 노드 반환
    HeapNode& getRight(int i) { return node[i * 2 + 1]; } // i번째 요소의 오른쪽 자식 노드 반환
    bool empty() { return size == 0; }
    bool full() { return size == MAX_SIZE - 1; }
    void print1(int i, int depth);
    void not_rotated(char** frame, int n, int depth, int center, int row);
    void print2();
    void H(char** H_tree, int n, int i, int j, int h, int v, int U, int D, int R, int L);
    int H_center(int n) { return n <= 1 ? 0 : 2 * H_center(n / 4) + 1; } // H-tree에서 중심 좌표를 구하는 함수
    int hor_depth(int n) { return n <= 7 ? 1 : 2 * hor_depth(n / 4); } // H-tree에서 가로 깊이를 구하는 함수
    int vert_depth(int n) { return n <= 15 ? 1 : 2 * hor_depth(n / 4); } // H-tree에서 세로 깊이를 구하는 함수
    void print3();

private:
    HeapNode node[MAX_SIZE];
    int size; // 힙에서 노드의 개수
};

```

<그림 6. HeapArray 클래스 선언>

(1). private 멤버 변수

```

private:
    HeapNode node[MAX_SIZE];
    int size; // 힙에서 노드의 개수
};

```

<그림 7. HeapArray 클래스의 private 멤버 변수>

위에서 만든 HeapNode 클래스 객체의 배열로 힙을 구현하였다. 배열엔 힙의 각 노드가 level-order순으로 들어간다. MAX_SIZE는 매크로 상수로 정의되어있고 값은 151이다. 과제에서 input 연산을 최대 150번 할 수 있다 하였으므로 힙의 노드의 수는 150을 넘지 않는다. 150보다 1 큰 151로 배열 크기를 잡은 이유는 배열의 0번째 요소를 사용하지 않기 때문이다. 이는 배열의 인덱스로 부모와 자식 노드를 찾기 위함이다.

변수 size는 힙에서 노드의 총 개수를 담는 정수 변수이다.

(2). 생성자

```

HeapArray() : size(0) {}

```

<그림 8. HeapArray 클래스 생성자>

클래스의 객체 생성 시 변수 size의 값을 0으로 초기화하여 객체를 생성한다.

(3). getParent(), getLeft(), getRight() 함수

```

HeapNode& getParent(int i) { return node[i / 2]; } // i번째 요소의 부모 노드 반환
HeapNode& getLeft(int i) { return node[i * 2]; } // i번째 요소의 왼쪽 자식 노드 반환
HeapNode& getRight(int i) { return node[i * 2 + 1]; } // i번째 요소의 오른쪽 자식 노드 반환

```

<그림 9. getParent(), getLeft(), getRight() 함수>

2. HeapArray 클래스

배열의 인덱스를 이용하여 해당 배열 요소의 각각 부모, 왼쪽 자식, 오른쪽 자식의 노드를 반환하는 함수이다. 배열의 0번째 요소를 제외하고 1번째 요소부터 level-order순으로 힙 노드가 들어가 있으므로 배열의 인덱스를 알면 부모와 자식 노드를 구할 수 있다. 힙이 complete binary tree이므로 i 번째 노드의 부모 노드는 $(i/2)$ 번째 노드이고, 왼쪽 자식 노드는 $(i*2)$ 번째, 오른쪽 자식 노드는 $(i*2+1)$ 번째 노드이다. 함수의 코드가 짧아 inline 함수로 정의하였고 반환된 값을 수정할 수 있도록 레퍼런스를 반환한다.

(4). empty(), full() 함수

```
bool empty() { return size == 0; }
bool full() { return size == MAX_SIZE - 1; }
```

<그림 10. empty(), full() 함수>

멤버 변수인 size의 값을 확인하여 배열이, 즉 힙이 비어있는지, 꽉 차있는지 확인하는 함수이다. full() 함수의 경우, MAX_SIZE가 151이므로 1을 뺀 값이 size와 같을 때 true를 반환한다.

(5). add() 함수

```
/* 힙에 새로운 노드를 추가 */
void HeapArray::add(char data)
{
    if (full())
        return;

    HeapNode new_node(data);
    int i = ++size; // 인덱스 변수

    while ((i != 1) && (new_node > getParent(i))) // 루트 아니고 부모 노드보다 크면
    {
        node[i] = getParent(i); // 부모 노드 끌어내림
        i /= 2; // 위로 이동
    }
    node[i].setData(data); // 최종 위치에 저장
}
```

<그림 11. add() 함수>

힙에 새로운 노드를 추가하는 함수이다. 인자로 추가할 노드의 data 값을 넣는다. 먼저, full() 함수를 이용하여 배열이 꽉 차있는지 확인한다. 꽉 차있지 않을 경우, 인자로 받은 data 값을 갖는 HeapNode 객체를 생성한다. 노드를 추가하므로 size의 값을 증가시키고 동시에 증가된 값을 인덱스 변수 i에 대입한다. 이걸 추가될 노드가 일단 힙의 끝 위치에 들어간다는 것을 의미한다. 그리고 i가 1이 아닐 때, 즉 힙이 비어있지 않았고, 힙의 끝 위치의 부모노드보다 새 노드의 값이 크다면 반복문에 들어가게 된다. 반복문에선 비교한 부모 노드를 아래로 끌어내리고 i에 2를 나눈 값을 넣어 새 노드가 들어갈 위치를 위로 올린다. 새 노드가 들어갈 위치의 부모 노드가 새 노드보다 크다면 그 위치가 새 노드의 적절한 위치이므로 반복문을 나와 그 위치에 data를 저장한다.

2. HeapArray 클래스

(6). remove() 함수

```

/* 힙에서 루트 노드를 제거 */
void HeapArray::remove()
{
    if (empty())
        return;

    HeapNode last = node[size]; // 마지막 노드
    int location = 1; // 초기 위치 루트
    int child = 2; // 왼쪽 자식 위치
    while (child <= size)
    {
        if ((child < size) && (getLeft(location) < getRight(location))) // 자식이 양 쪽 다 있고 오른쪽 자식이 더 크다면
            child++; // 오른쪽 자식과 비교
        if ((last > node[child]) || (last.getData() == node[child].getData())) // 마지막 노드 값이 자식보다 크면
            break; // 이동 완료
        else // 한 단계 아래로 이동
        {
            node[location] = node[child];
            location = child;
            child *= 2;
        }
    }
    node[location] = last; // 최종 위치에 저장
    size--; // 노드 개수 감소
}

```

<그림 12. remove() 함수>

힙의 루트 노드를 제거하는 함수이다. 힙이 비어있을 땐 함수가 바로 종료된다. 루트 노드의 제거는 힙의 마지막 노드가 루트로 올라간 다음 적절한 위치를 찾아 아래로 내려가는 방식이므로 마지막 노드를 last 객체에 저장해둔다. location은 마지막 노드가 저장될 위치 변수이고 child 변수는 location 위치의 왼쪽 자식 위치를 저장하는 변수다. 처음엔 루트 노드의 위치로 가므로 location은 1, child는 2로 초기화하였다. 그리고 난 뒤, 힙의 size 범위 내에서 자식 노드의 data와 마지막 노드의 data를 비교하는 작업을 한다. 먼저, 왼쪽 자식과 오른쪽 자식을 비교하여 더 큰 자식을 찾아내고 그 자식과 마지막 노드를 비교한다. 비교 결과, 마지막 노드가 자식보다 작으면 location과 child 변수의 값을 바꿔 마지막 노드가 들어갈 위치를 한 단계 내리고, 자식은 위로 올린다. 그렇지 않다면 그 위치가 적절한 위치이므로 반복문을 나와 location을 인덱스로 한 요소에 마지막 노드를 저장한다. 마지막으로 size 값을 하나 줄여 마지막 요소를 지워준다.

(7). print1() 함수

```

/* rotated form 출력 함수 */
void HeapArray::print1(int i, int depth)
{
    if (i <= size)
    {
        /* Backward In-Order Traversal */
        print1(i * 2 + 1, depth + 1); // 오른쪽 자식
        cout << setw(2 * depth) << node[i].getData() << endl; // 자기자신
        print1(i * 2, depth + 1); // 왼쪽 자식
    }
}

```

<그림 13. print1() 함수>

힙의 rotated form을 출력하는 함수이다. 호출 시 힙 배열의 i번째 요소부터 끝 요소까지로 구성된 90도 회전 트리가 출력된다. 힙을 전체적으로 출력하려면 인자 i에 1을 넣어주면 된다. 두 번째 인자 depth는 트리에서 부모와 자식 사이 간격 너비이다. 반시계 방향으로 90도 회전된 트리이므로 첫 줄에는 오른쪽 자식이, 그 다음 줄엔 자기 자신, 그 다음 줄엔 왼쪽 자식이 출력돼야 한다. 그렇기 때문에 backward in-order 순으로 출력이 되고 자식 부분은 함수를 재귀 호출하여 가장 깊은 노드까지 재귀적으로 출력되게 하였다.

2. HeapArray 클래스

(8). not_rotated(), print2() 함수

```
/* not-rotated form을 위한 재귀 함수 */
void HeapArray::not_rotated(char** frame, int n, int depth, int center, int row)
{
    if (n > size)
        return;
    frame[depth][center] = node[n].getData(); // 자기 자신
    if (2 * n <= size) // 왼쪽 자식
        not_rotated(frame, 2 * n, depth + 1, center - (int)pow(2, row - (depth + 2)), row);
    if (2 * n + 1 <= size) // 오른쪽 자식
        not_rotated(frame, 2 * n + 1, depth + 1, center + (int)pow(2, row - (depth + 2)), row);
}
```

<그림 14. not_rotated() 함수>

not_rotated() 함수는 not-rotated form 트리 출력을 위해 인자로 받은 frame 2차원 배열에 노드의 data 값을 채우는 함수이다. 두 번째 인자 n과 세 번째 인자 depth는 각각 출력을 시작할 노드의 인덱스와 깊이 값이다. 트리 전체 출력을 위해선 1과 0을 넣으면 된다. 인자 center는 2차원 배열 frame 한 행의 가운데 인덱스 값이고 마지막 인자 row는 이차원 배열 frame의 총 행의 개수이다.

함수 내부는 재귀적으로 구성되어 있다. n이 size보다 클 경우, 즉 힙의 범위를 벗어날 경우 함수는 종료된다. 인자로 받은 depth와 center의 위치에 n번째 노드 data가 들어가고 왼쪽 자식, 오른쪽 자식의 경우 각각 n, depth, center의 값을 바꾸어 not_rotated 함수를 재귀 호출한다. 이렇게 하여 가장 깊은 노드까지 2차원 배열에 담기게 된다.

```
/* 2차원 배열을 이용한 not-rotated form 출력 */
void HeapArray::print2()
{
    int row = (int)(log(size) / log(2)) + 1; // 행의 개수
    int col = (int)pow(2, row) - 1; // 열의 개수
    int center = col / 2; // 한 행의 중심

    /* 배열 크기를 변수로 받기 위해 동적 할당 */
    char** frame = new char*[row];
    for (int i = 0; i < row; i++)
    {
        frame[i] = new char[col];
        memset(frame[i], ' ', col);
    }

    /* 배열 채운 후 출력 */
    not_rotated(frame, 1, 0, center, row);
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
            cout << frame[i][j];
        cout << endl;
    }

    /* 동적 할당 해제 */
    for (int i = 0; i < row; i++)
        delete[] frame[i];
    delete[] frame;
}
```

<그림 15. print2() 함수>

print2() 함수는 2차원 배열을 동적 할당하고 not_rotated() 함수를 호출하여 배열을 채운 뒤 출력하는 함수이다. 힙의 size 값을 이용하여 변수 row, col 값을 알맞게 설정한 뒤 이 두 변수의 크기를 갖는 2차원 배열을 동적 할당한다. 동적 할당과 동시에 전체 배열 내용을 공백 문자로 채워주었다. 그리고 동적 할당한 배열을 인자로 넣은 not_rotated 함수를 호출하여 배열을 채운다. 이때, center의 값은 col을 2로 나눈 정수 값이다. 배열을 출력한 한 뒤 동적 할당한 메모리를 해제한다.

2. HeapArray 클래스

(9). H(), H_center(), hor_depth(), vert_depth(), print3() 함수

```
int H_center(int n) { return n <= 1 ? 0 : 2 * H_center(n / 4) + 1; } // H-tree에서 중심 좌표를 구하는 함수
int hor_depth(int n) { return n <= 7 ? 1 : 2 * hor_depth(n / 4); } // H-tree에서 가로 깊이를 구하는 함수
int vert_depth(int n) { return n <= 15 ? 1 : 2 * hor_depth(n / 4); } // H-tree에서 세로 깊이를 구하는 함수
```

<그림 16. H_center(), hor_depth(), vert_depth() 함수>

H-tree를 구성하고 출력하는 H() 함수와 print3() 함수에 앞서 이 두 함수 호출을 위해 필요한 H_center(), hor_depth(), vert_depth() 함수를 설명하겠다.

H_center 함수는 힙의 size를 인자로 넣으면 H-tree에서 중심 좌표를 구해주는 함수이다. H-tree도 2차원 배열에 data 값을 저장하여 출력하는 방식인데, 여기서 2차원 배열은 행과 열의 수가 같은 정사각형 모양이다. 즉, H_center로 반환된 값을 x, y값으로 하는 좌표가 H-tree의 중심이 되는 셈이다.

hor_depth() 함수는 H-tree에서 가로 깊이, 즉 가로선 상에 위치한 부모와 자식 노드 사이의 필요한 간격을 반환하는 함수이다. vert_depth() 함수는 이와 반대로 세로선 상에 위치한 부모와 자식 노드 사이의 필요한 간격을 반환하는 함수이다.

```
/* H-tree form을 위한 재귀 함수 */
void HeapArray::H(char** H_tree, int n, int i, int j, int h, int v, int U, int D, int R, int L)
{
    if (n > size)
        return;
    H_tree[i][j] = node[n].getData(); //자기자신
    if (2 * n <= size) // 왼쪽 자식의 경우
    {
        for (int index = 1; index < h; index++) // 왼쪽으로 #
            H_tree[i + index*V[L][0]][j + index*V[L][1]] = '#';
        for (int index = 1; index < v; index++) // 왼쪽 위아래로 #
        {
            H_tree[i + v * V[L][0] + index * V[U][0]][j + v * V[L][1] + index * V[U][1]] = '#';
            H_tree[i + v * V[L][0] + index * V[D][0]][j + v * V[L][1] + index * V[D][1]] = '#';
        }
        H_tree[i + h*V[L][0]][j + h*V[L][1]] = node[2 * n].getData(); // 왼쪽 자식 저장
        H(H_tree, 4 * n, i + v*(V[L][0] + V[U][0]), j + h*(V[L][1] + V[U][1]), h / 2, v / 2, D, U, L, R); // 왼쪽 위로 이동
        H(H_tree, 4 * n + 1, i + v*(V[L][0] + V[D][0]), j + h*(V[L][1] + V[D][1]), h / 2, v / 2, U, D, R, L); // 왼쪽 아래로 이동
    }
    if (2 * n + 1 <= size) // 오른쪽 자식의 경우
    {
        for (int index = 1; index < h; index++) // 오른쪽으로 #
            H_tree[i + index*V[R][0]][j + index*V[R][1]] = '#';
        for (int index = 1; index < v; index++) // 오른쪽 위아래로 #
        {
            H_tree[i + v * V[R][0] + index * V[U][0]][j + v * V[R][1] + index * V[U][1]] = '#';
            H_tree[i + v * V[R][0] + index * V[D][0]][j + v * V[R][1] + index * V[D][1]] = '#';
        }
        H_tree[i + h*V[R][0]][j + h*V[R][1]] = node[2 * n + 1].getData(); // 오른쪽 자식 저장
        H(H_tree, 4 * n + 2, i + v*(V[R][0] + V[D][0]), j + h*(V[R][1] + V[D][1]), h / 2, v / 2, U, D, R, L); // 오른쪽 아래로 이동
        H(H_tree, 4 * n + 3, i + v*(V[R][0] + V[U][0]), j + h*(V[R][1] + V[U][1]), h / 2, v / 2, D, U, L, R); // 오른쪽 위로 이동
    }
}
```

<그림 17. H() 함수>

H() 함수는 첫 번째 인자로 전달받은 2차원 배열에 H-tree form으로 각 노드의 data를 저장하는 함수이다. 이 함수는 처음 호출됐을 때 자기 자신과 왼쪽 자식, 오른쪽 자식 노드의 data까지 저장하고 나머지는 재귀적으로 함수가 호출되어 전체적으로 H-tree form을 형성한다. 두 번째 인자 n에는 힙의 size가 들어가서 힙 범위를 넘어가면 함수가 종료되고, i와 j는 첫 요소(루트 노드)가 저장될 배열의 좌표를 의미한다. h와 v는 각각 가로, 세로 부모와 자식 노드 사이 간격을 의미하고 U, D, R, L은 전역으로 선언된 원소 벡터를 의미하는 배열 V의 행 인덱스를 의미한다. 코드 사이에 반복문을 이용하여 자식 data가 저장되는 위치를 추적해 그 길을 #으로 채우는 부분도 추가하였다.

2. HeapArray 클래스

```

/* 2차원 배열을 이용한 H-tree form 출력 */
void HeapArray::print3()
{
    int side = H_center(size) * 2 + 1; // 배열 한 변의 길이

    /* 배열 크기를 변수로 받기 위해 동적 할당 */
    char** H_tree = new char*[side];
    for (int i = 0; i < side; i++)
    {
        H_tree[i] = new char[side];
        memset(H_tree[i], ' ', side);
    }

    /* 배열 채운 후 출력 */
    H(H_tree, 1, H_center(size), H_center(size), hor_depth(size), vert_depth(size), 0, 1, 2, 3);
    for (int i = 0; i < side; i++)
    {
        for (int j = 0; j < side; j++)
            cout << H_tree[i][j];
        cout << endl;
    }

    /* 동적 할당 해제 */
    for (int i = 0; i < side; i++)
        delete[] H_tree[i];
    delete[] H_tree;
}

```

<그림 18. print3() 함수>

print3() 함수는 2차원 배열을 할당하여 H() 함수로 배열을 채운 뒤 이를 출력하는 함수이다. 배열은 행과 열의 개수가 같은 정사각모양의 배열이며 한 행의 길이는 H_center() 함수로 얻어낸 값에 2배를 한 뒤 1을 더한 값으로 설정하였다. 2차원 배열을 동적 할당함과 동시에 배열을 우선 공백 문자로 채우고 이 배열을 H() 함수 인자로 넣어 배열을 완성하였다. 이를 화면에 출력하고 나서 동적 할당된 메모리를 해제한다.

3. main 함수

```
ifstream inStream("input.txt"); // 파일 입력
if (inStream.fail()) // 파일 열기 실패 시 종료
{
    cout << "Input file opening failed.\n";
    exit (1);
}

int size; // 연산 횟수
string operations; // 연산 문자열

inStream >> size; // 첫 번째 줄 입력

if (size > 150) // 범위 초과 시
    size = 150; // 150 저장

while (!inStream.eof()) // 파일의 끝까지
{
    /* 공백 문자를 제거한 나머지 문자를 문자열에 저장*/
    string temp;
    inStream >> temp;
    operations += temp;
}
```

<그림 19. main 함수의 파일 입출력 부분>

힙 연산이 들어가 있는 input.txt 파일 입력스트림 객체를 생성한다. 파일 열기 실패 시 오류 메시지를 출력하며 프로그램이 종료되도록 하였다. 파일 첫 줄의 연산 횟수 정수 값을 받기 위해 size 변수를 선언하였고, 두 번째 줄의 힙 연산 문자열을 받기 위해 operations string 객체를 생성하였다. 파일 입력스트림으로 첫 줄을 받아 이를 int 변수 size에 저장하고 이 값이 연산 최대 범위인 150을 넘어갈 경우 150으로 값을 맞추었다. 이 후 파일의 나머지 내용을 받기 위해 파일의 끝에 도달할 때까지 반복문을 돌리고 반복문 내에서는 파일 입력스트림으로 문자열을 받아 operations에 덧붙이는 방식으로 공백문자를 제거하여 남은 내용을 저장하게 하였다.

```
HeapArray myHeap; // 힙 생성
int i = 0; // 문자열 인덱스
for (int n = 0; n < size; n++) // 파일의 정수 값만큼 연산
{
    if (operations[i] == 'I') // insert
    {
        myHeap.add(operations[i + 1]);
        i += 2;
    }
    else if (operations[i] == 'D') // delete
    {
        myHeap.remove();
        i++;
    }
}
```

<그림 20. 힙 생성 및 구성>

HeapArray 객체로 힙을 생성하고 문자열 operations에 들어있는 문자를 확인해 이 힙을 채운다. 변수 i는 operations 문자열의 첫 문자부터 끝 문자까지 확인할 인덱스 변수이고, 변수 n은 힙 연산의 횟수를 세는 변수이다. operations의 문자를 확인하여 'I'라면 그 다음 문자를 힙에 insert 하고 i 값을 2 증가시킨다. 문자가 'D'라면 힙의 루트 노드를 제거하고 i를 1 증가시킨다. 이를 파일 첫 줄에서 받은 size 횟수만큼 반복한다.

3. main 함수

```
/* 출력 */
cout << "1. rotated form\n" << endl;
myHeap.print1(1, 1);
cout << endl << "=====" << endl << endl;

cout << "2. not-rotated form\n" << endl;
myHeap.print2();
cout << endl << "=====" << endl << endl;

cout << "3. H-tree form\n" << endl;
myHeap.print3();
cout << endl;

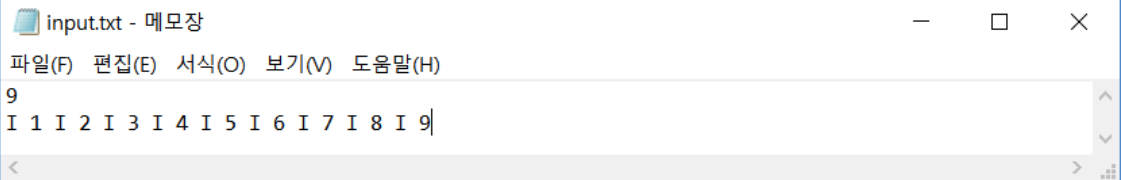
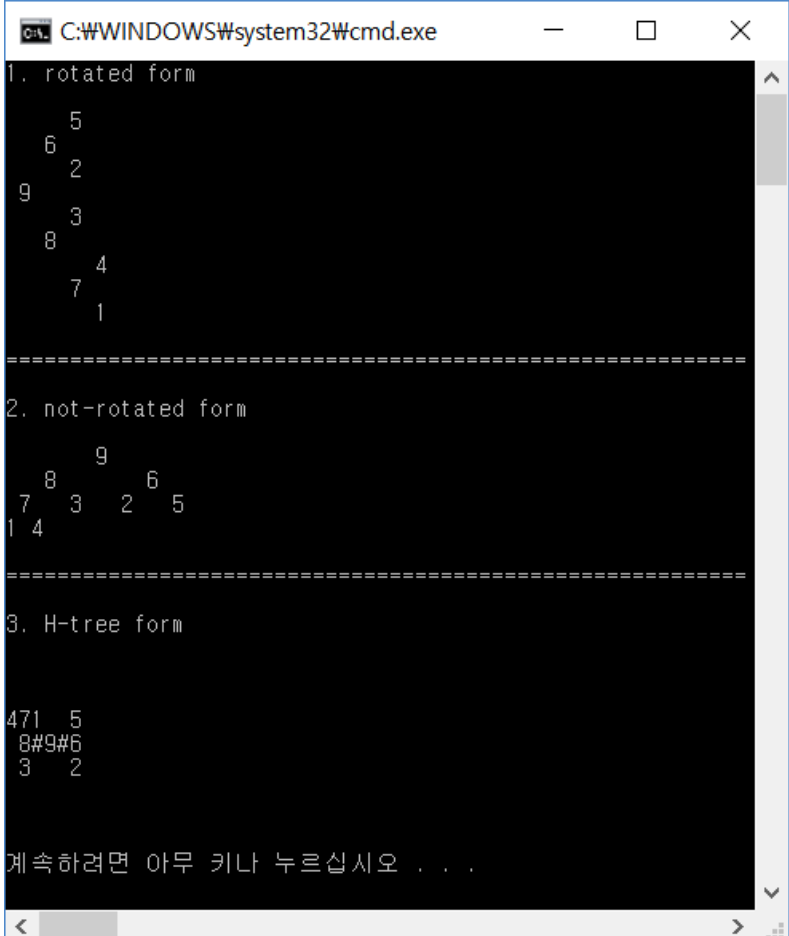
inStream.close();

return 0;
```

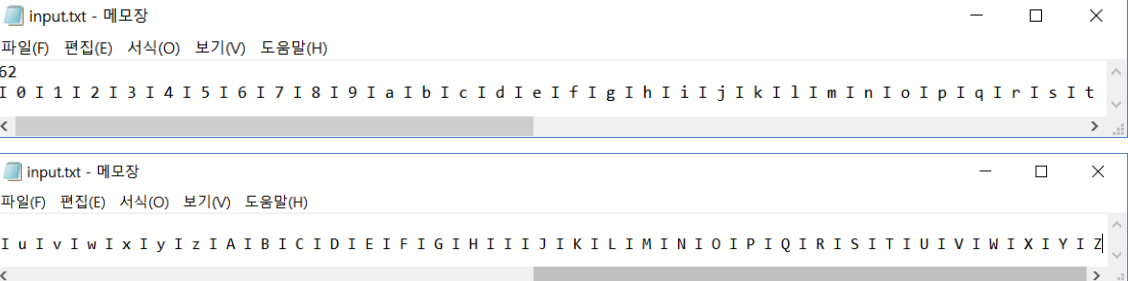
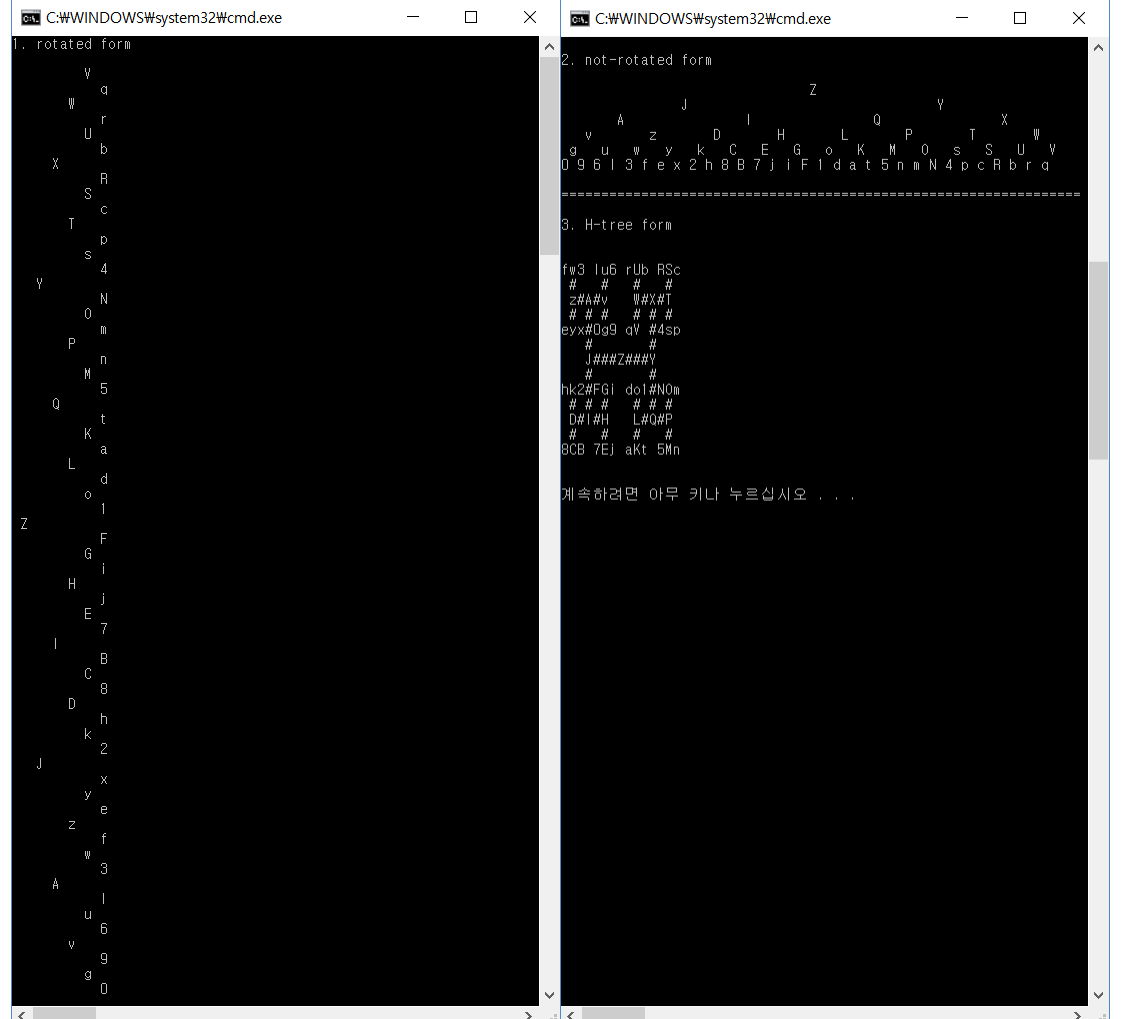
<그림 21. 힙 트리 출력>

HeapArray 클래스의 함수 print1(), print2(), print3()를 각각 호출하여 rotated form, not-rotated form, H-tree from 트리를 차례로 출력한다. 마지막으로 열었던 파일을 닫아주며 프로그램을 종료한다.

4. 실행 결과

	input.txt	
1	실행 결과	

4. 실행 결과

	input.txt	 <p>input.txt - 메모장 파일(F) 편집(E) 서식(O) 보기(V) 도움말(H) 62 I 0 I I I 2 I 3 I 4 I 5 I 6 I 7 I 8 I 9 I a I b I c I d I e I f I g I h I i I j I k I l I m I n I o I p I q I r I s I t I u I v I w I x I y I z I A I B I C I D I E I F I G I H I I I J I K I L I M I N I O I P I Q I R I S I T I U I V I W I X I Y I Z</p>
2	실행 결과	 <p>C:\WINDOWS\system32\cmd.exe 1. rotated form v w u x s t p s 4 y n o m p n m 5 q t k a l d o 1 g i h j e 7 i b c 8 d h k 2 j x y e z f w 3 a l u 6 v g g 0</p> <p>C:\WINDOWS\system32\cmd.exe 2. not-rotated form J Z v u w y k c e h g o k m p s t s u v 0 9 6 l 3 f e x 2 h 8 B 7 j i f l d a t 5 n m N 4 p c R b r q</p> <p>3. H-tree form fw3 lu6 rUb RSc # # # z##v ##X#T # # # eyx#0g9 qV #4sp # J###Z###V # hk2#FGi do1#N0m # # # D#l#H L#Q#P # # # 8CB 7Ej aKt 5Mn 계속하려면 아무 키나 누르십시오 . . .</p>

3

5. 고 찰

본 과제를 시작하기에 앞서 어떻게 접근할지부터 난관이었다. 먼저 힙을 구성해야하는데 STL을 이용하려 했더니 더 복잡하기만 하였다. 그래서 직접 힙을 구현하는 것이 더 유연하겠다는 생각이 들었고 처음엔 Linked List로 시도해보았다. 그런데 Linked List로 하려하니 새 노드를 추가하고 옮기는 게 쉽지 않았다. 새 노드를 추가하려면 마지막 위치를 먼저 알아내야 하는데 방법이 잘 떠오르지 않았다. 결국 인터넷 검색을 통해 Linked List보다 배열을 이용하는 것이 더 수월하다는 글을 보고 배열을 이용하기로 하였다. 어차피 연산 횟수 범위가 정해져있기 때문에 정적 배열을 이용해도 문제가 없다 판단하였다. 수업 시간에 잠깐 언급하고 넘어갔었던 배열을 이용한 부모, 자식 노드를 구하는 방법을 떠올려 함수를 만들었다. 그리고 이를 이용하여 시간이 좀 걸렸지만 새 노드를 추가하고, 루트 노드를 제거하는 함수도 만들었다. 처음엔 이 두 함수를 일일이 노드들을 swap해가며 적절한 위치로 가도록 짰었는데, 위치 변수를 만들어 이를 이용한 몇몇 코드를 보고 힌트를 얻어서 최종 위치를 구한 후 그 위치에 노드를 놓는 방법으로 수정하였다.

이렇게 힙을 구성하고 난 뒤 클래스의 함수들이 정상적으로 작동하는지 테스트가 필요했다. 그러기 위해선 input.txt 파일을 읽어 오는 것이 선행이 되어야했다. 지난 학기 C++ 수업 때 파일 입출력과 String 클래스 부분을 건너뛰었기 때문에 다시 C++ 책을 펴 이 부분에 대해 공부를 하였다. 다행히도 여러 시도를 거친 결과, 오랜 시간 걸리지 않고 파일을 잘 읽어 변수에 저장할 수 있었다. >> redirection 연산을 이용해 파일의 내용을 받았는데 이는 공백 문자까지만 입력을 받는 연산자였다. 하지만 이에 대해 잘 모르고 전체 내용이 입력이 안 되자 임시 객체 temp를 만들어 이를 덧붙이는 방법을 사용했다. 이렇게 하니 최종 문자열엔 공백 문자가 제거되어 들어가서 이후 작업이 더 수월했다. 이 문자열에 문자 l가 들어오면 add() 함수를, D가 들어오면 remove() 함수를 호출시키고 지금까지의 과정을 확인하기 위해 간단히 정렬 없이 배열 내용을 출력하는 함수를 만들어 호출해보았다. 그런데, 아스키 코드 순서와 과제의 문자 크기순서가 달라 영어 소문자가 더 높은 위치에 가게 되었다. 이를 어떻게 해결할까 하다 비교 연산자를 오버로딩 하면 되겠다는 생각이 들었다. 그래서 각 힙 노드를 뜻하는 클래스를 만들어 그 안에 영문자 기준, 결과를 반대로 반환하는 비교 연산자 오버로딩 함수를 넣어주었다. 이상이 없는 것을 확인하고 rotated-form으로 출력하는 함수부터 접근하였다.

강의 자료에 있던 함수는 Linked List로 구현한 힙 기준으로 작성된 거라 약간의 수정이 필요했다. 힙 클래스 내에 양 쪽 자식 노드를 반환하는 함수를 미리 구현해놓았기 때문에 크게 어렵지 않았다. 문제는 그 다음부터였다. not-rotated form을 출력해야하는데 어떤 식으로 출력해야하는지 도무지 감이 잡히지 않았다. 결국 pseudo-code가 주어진 H-tree form부터 해보기로 했다. 처음엔 pseudo-code조차 이해되지 않았지만 무작정 pseudo-code를 Visual Studio에 옮겨 오류가 나는 부분을 하나씩 수정하였다. 이렇게 하다 보니 각 줄이 의미하는 바를 조금씩 알게 되었다. 이렇게 완성한 H-tree 함수를 다양한 입력 값을 넣어 호출해보는데 과제에서 예시로 나온 실행결과와 조금 달랐다. 예시 실행결과에 맞추기 위해 어쩔 수 없이 코드 수정 및 새 함수 추가가 필요했고 워낙 기존 코드가 복잡했던 터라 꽤나 골머리를 앓았다. 추가로 #도 출력해야했기 때문에 이 부분에서 가장 많은 시간이 걸렸다. 힘들게 H-tree를 구현하고 나니 not-rotated form은 어떻게 접근해야할지 바로 보였다. H-tree에서의 방법을 조금 수정하였더니 재귀 방법으로 쉽게 구현할 수 있었다.

워낙 짜야 할 것들이 많다보니 하나하나가 산 넘어 산이었다. 특히 H-tree의 재귀에 대한 부분과 2차원 배열에서의 이동 부분에서 많은 생각과 시행착오 과정이 필요했다. 그래도 이 모든 것을 해결하고 나니 평소 이론으로만 접하고 사용할 기회가 없었던 Recursion을 직접 사용해보고 이해하고 조금 익숙해지는 기회가 된 것 같아 뜻 깊었다.