Q20

此題需要比較Min heap 與leftist heap 的效能

Min heap

Define node

```
typedef struct Heap
{
    int *data;    // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int size;     // Current number of elements in min heap
} Heap;
```

初始化heap

先malloc給它應有的空間

開給他data的array 大小為MAX_N

第一個陣列當作data取完的標示 也讓root在data[1]的位子

時間複雜度為O(n)

```
Heap *create()
{
    Heap *heap = (Heap *)malloc(sizeof(Heap));
    heap->data = (int *)malloc(sizeof(int) * MAX_N);
    heap->capacity = MAX_N;
    heap->size = 0;
    heap->data[0] = -1; //to know when the array used up
    return heap;
}
```

insert heap

插入一個node的時間複雜度度O(log_2n)

因為tree的高度為h node數為$2^h - 1$

最多swap h次

```
void insert(Heap *heap, int val)
{
    if (heap->size == heap->capacity - 1)
        return; //out of capacity

    int index = ++heap->size; //the position to be place
    while (val < heap->data[parent(index)])
    {
        heap->data[index] = heap->data[parent(index)]; //swap down the parent
        index = parent(index);                         //make the index be the parent index
    }
    heap->data[index] = val;
}
```

delete min

時間複雑度O($log_2 n$)

```
int pop_min(Heap *heap)
{
    int index = 1;
    int top = heap->data[1];
    int last = heap->data[heap->size--];
    while (left(index) <= heap->size)
    {
        if (right(index <= heap->size) && (last > heap->data[right(index)] || last > heap->data[left(index)]))
        {
            if (heap->data[right(index)] < heap->data[left(index)]) //deter which child to change
            {
                heap->data[index] = heap->data[right(index)];
                index = right(index);
            }
            else
            {
                heap->data[index] = heap->data[left(index)];
                index = left(index);
            }
        }
        else if (last > heap->data[left(index)])
        {
            heap->data[index] = heap->data[left(index)];
            index = left(index);
        }
        else
            break;
    }
    heap->data[index] = last;
    return top;
}
```

leftist heap

為什麼要提出left heap 這麼一個概念?

min heap數據結構在支持合併（merge）操作的時候是比較差的。比如對於一個二叉查找樹來說，要把兩個二叉查找樹合併，那麼可能需要把一個二叉樹的結點一個一個的插入到另一個二叉樹中，這個的時間消耗是一個很恐怖的事情。因此我們引入left heap。

left的存在就是為了合併（merge），因此主要關注的就是怎樣把兩個左堆合併。

define node

```
typedef struct node {
    int val, npl;
    struct node *left, *right;
} Node;
```

初始

```
Node* init(int x)
{
    Node* tmp = (Node*)malloc(sizeof(Node));
    tmp->val = x;
    tmp->npl = 0;
    tmp->left = tmp->right = NULL;
    return tmp;
}
```

merge 合併

```
Merge(LeftistNode * h1, LeftistNode * h2)
{
    if (h1 == NULL)
        return h2;
    if (h2 == NULL)
        return h1;
    if (h1->element < h2->element)
        return Merge1(h1, h2);
    else
        return Merge1(h2, h1);
}
Merge1(LeftistNode * h1,LeftistNode * h2)
{
    if (h1->left == NULL)
        h1->left = h2;
    else
    {
        h1->right = Merge(h1->right, h2);
        if (h1->left->dist < h1->right->dist)
            swapChildren(h1);
        h1->dist = h1->right->dist + 1;
```

```
    }
    return h1;
}
```

leftist 的時間複雜度

Delete Min: O(Log n) [same as both Binary and Binomial]
Insert: O(Log n) [O(Log n) in Binary and O(1) in
Binomial and O(Log n) for worst case]
Merge: O(Log n) [O(Log n) in Binomial]