F74109016 葉惟欣 資訊系 HW21

程式碼

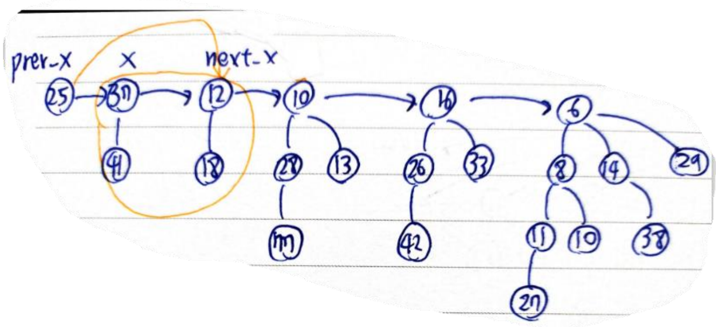
```c
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
#include<time.h>
/***********binomial heap****************/
struct node {
    int n;
    int degree;
    struct node* parent;
    struct node* child;
    struct node* sibling;
};

struct node* MAKE_bin_HEAP();
int bin_LINK(struct node*, struct node*);
struct node* CREATE_NODE(int);
struct node* bin_HEAP_UNION(struct node*, struct node*);
struct node* bin_HEAP_INSERT(struct node*, struct node*);
struct node* bin_HEAP_MERGE(struct node*, struct node*);
struct node* bin_HEAP_EXTRACT_MIN(struct node*);
int REVERT_LIST(struct node*);

int count = 1;

struct node* MAKE_bin_HEAP() {    //做出一個空的heap
    struct node* np;
    np = NULL;
    return np;
}

struct node * H = NULL;
struct node * Hr = NULL;
```

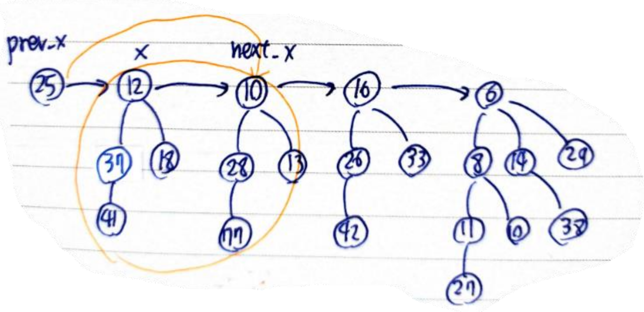
H 是會指向 binomial heap 的 minimum

```c

int bin_LINK(struct node* y, struct node* z) {    //有heap的link
    y->parent = z;
    y->sibling = z->child;
    z->child = y;
    z->degree = z->degree + 1;
}

struct node* CREATE_NODE(int k) {            //新增一個節點
    struct node* p;//new node;
    p = (struct node*) malloc(sizeof(struct node));
    p->n = k;
    return p;
}

```

Bin_LINK 函數是 將 y 做為 z 的 parent，在合併的時候會用到。

將值為 12 的節點變成 37 的 parent。(如圖)





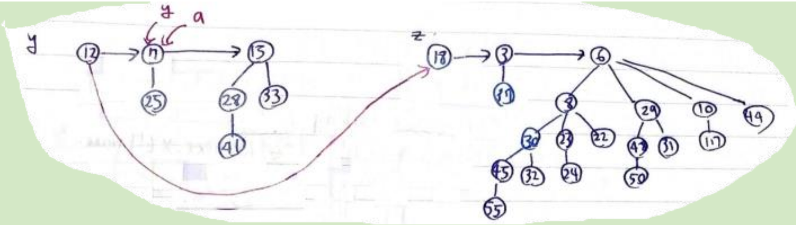
```c
struct node* bin_HEAP_UNION(struct node* H1, struct node* H2) { //將兩個heap結合
    struct node* prev_x;
    struct node* next_x;
    struct node* x;
    struct node* H = MAKE_bin_HEAP(); //先建一個空的heap這是作為合併後的heap
    H = bin_HEAP_MERGE(H1, H2);
    if (H == NULL)
        return H;
    prev_x = NULL;
    x = H;
    next_x = x->sibling;
    while (next_x != NULL) {          //如果有兄弟節點
        if ((x->degree != next_x->degree) || ((next_x->sibling != NULL) && (next_x->sibling)->degree == x->degree)) {
            //如果x的degree不等於x兄弟的degree，或是x兄弟的兄弟degree等於x的degree。
            prev_x = x;
            x = next_x;
        }
        else {
            if (x->n <= next_x->n) { //大小
                x->sibling = next_x->sibling;
                bin_LINK(next_x, x); //x是父節點，next_x是子節點
            }
            else {
                if (prev_x == NULL){
                    H = next_x;
                }
                else{
                    prev_x->sibling = next_x;
                }
                bin_LINK(x, next_x);
                x = next_x;
            }
        }
        next_x = x->sibling;
    }
    return H;
}

struct node* bin_HEAP_INSERT(struct node* H, struct node* x) {
    struct node* H1 = MAKE_bin_HEAP();
    x->parent = NULL;
    x->child = NULL;
    x->sibling = NULL;
    x->degree = 0;
    H1 = x;
    H = bin_HEAP_UNION(H, H1);
    return H;
}
```

Step1： 先做只有一個節點 x 的 heap H’
Step2 : Binomial-Heap-Union(H,H’)再將兩個節點做聯集。

```c
struct node* bin_HEAP_MERGE(struct node* H1, struct node* H2) {
    struct node* H = MAKE_bin_HEAP();
    struct node* y;
    struct node* z;
    struct node* a;
    struct node* b;
    y = H1;
    z = H2;
    if (y != NULL) {
        if (z != NULL && y->degree <= z->degree)
            H = y;
        else if (z != NULL && y->degree > z->degree)
            H = z;
        else //z 為Null
            H = y;
    }
    else //y是null
        H = z;
    while (y != NULL && z != NULL) {
        if (y->degree < z->degree) {
            y = y->sibling;
        } else if (y->degree == z->degree) {
            a = y->sibling;
            y->sibling = z;
            y = a;
        } else {
            b = z->sibling;
            z->sibling = y;
            z = b;
        }
    }
    return H;
}
```

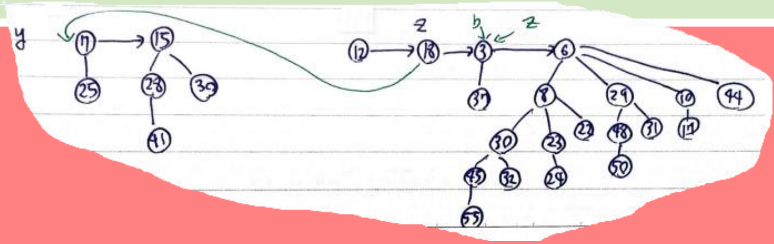
bin_HEAP_UNION 的函數 先把兩個 HEAP 用 bin_HEAP_MERGE 合併。如果是插入的話，就先將欲插入的節點視為單獨的 HEAP 再與原先的 HEAP 做合併。合併完再做調整。

合併的舉例:



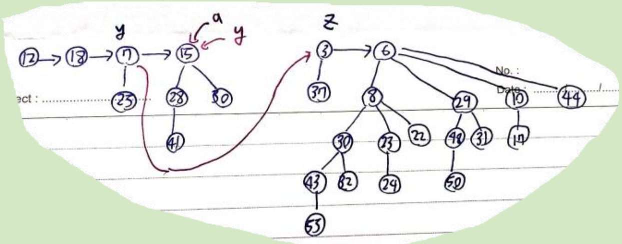
```
else if (y->degree == z->degree) {
  a = y->sibling;
  y->sibling = z;
  y = a;
```

當 y 的 degree 跟 z 的 degree 相等，就將 y 往後移一個，原先的 y 連到 z 前面

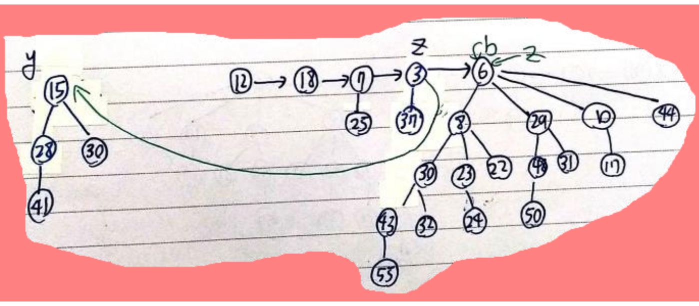


```
else {
  b = z->sibling;
  z->sibling = y;
  z = b;
```

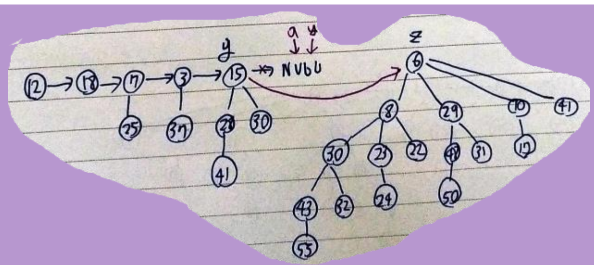
當 y 的 degree 大於 z 的 degree，將 z 往後移一個，原先的 z 連到 y 前面。



```
else if (y->degree == z->degree) {
  a = y->sibling;
  y->sibling = z;
  y = a;
```

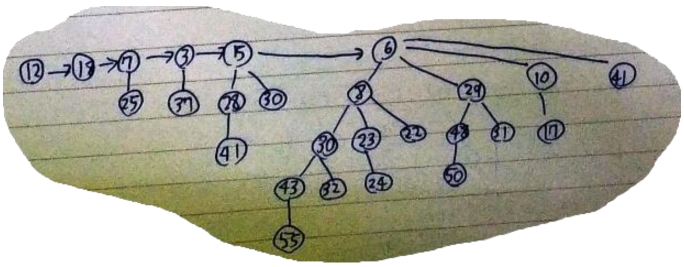


```
else {
  b = z->sibling;
  z->sibling = y;
  z = b;
```

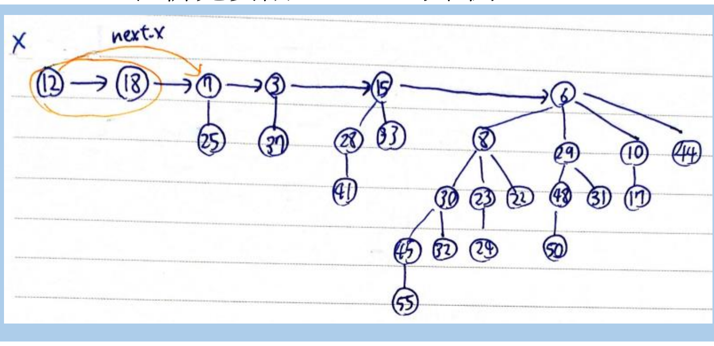


```
if (y->degree < z->degree) {
    y = y->sibling;
```

當 y 的 degree 小於 z 的 degree，將 y 連到 z 前面。遞迴結束

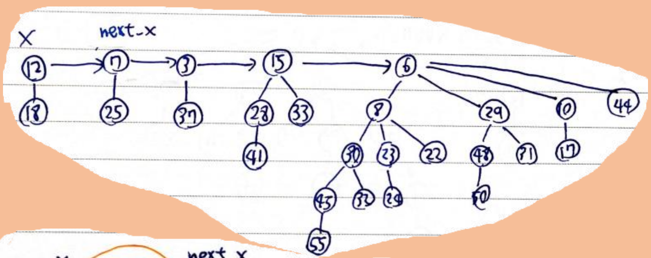


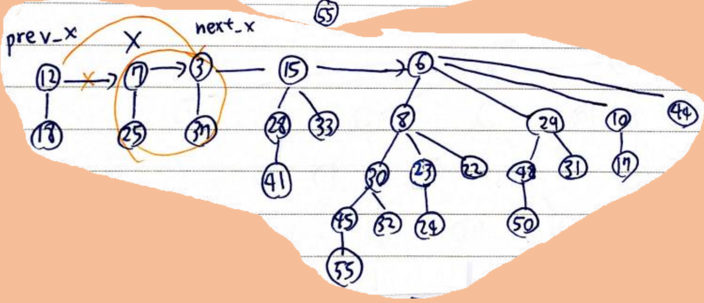
合併完要做 **UNION** 的舉例




```
else {
    if (x->n <= next_x->n) { //大小
        x->sibling = next_x->sibling;
        bin_LINK(next_x, x); //x是父節點，next_x是子節點
    }
```

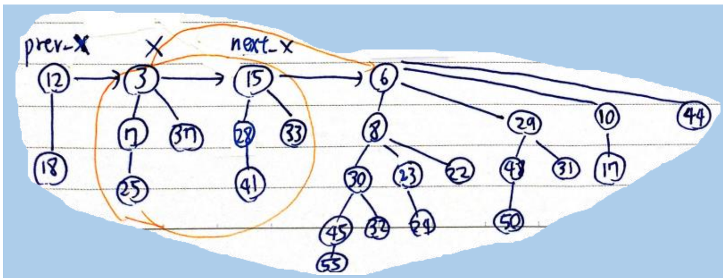
x 的 degree 等於 x 兄弟的 degree 但不等於 x 兄弟的兄弟的 degree，x 的值小於兄弟的值



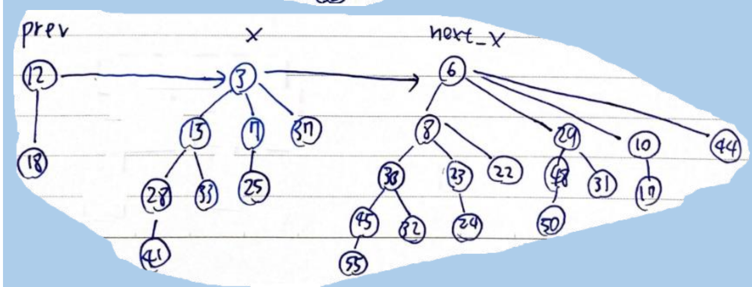


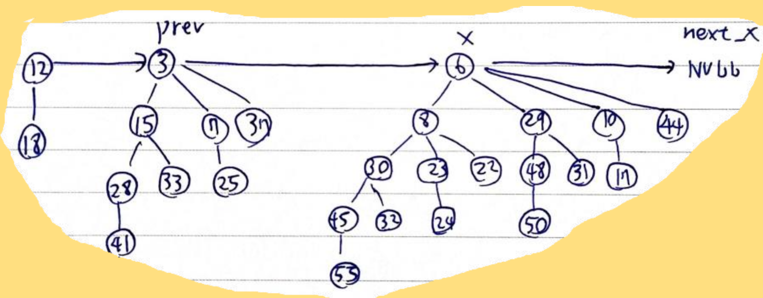

```
else {
    if (prev_x == NULL){
        H = next_x;
    }
    else{
        prev_x->sibling = next_x;
    }
    bin_LINK(x, next_x);
    x = next_x;
}
```

x 的 degree 等於 x 兄弟的 degree 但等於 x 兄弟的兄弟的 degree，x 的值大於兄弟的值




```
else {
    if (x->n <= next_x->n) { //大小
        x->sibling = next_x->sibling;
        bin_LINK(next_x, x); //x是父節點，next_x是子節點
    }
```





當 x 的 degree 不等於兄弟的 degree，(此例子符合)或是，x 的 degree 等於兄弟的兄弟的 degree

➔所有指標往前移一個

```
if ((x->degree != next_x->degree) || ((next_x->sibling != NULL) && (next_x->sibling)->degree == x->degree)) {
    //如果x的degree不等於x兄弟的degree，或是x兄弟的兄弟degree等於x的degree。
    prev_x = x;
    x = next_x;
}
```

```c
struct node* bin_HEAP_EXTRACT_MIN(struct node* H1) {
    int min;
    struct node* t = NULL;
    struct node* x = H1;
    struct node *Hr;
    struct node* p;
    Hr = NULL;
    if (x == NULL) {
        return x;
    }
    min=x->n;
    p = x;
    while (p->sibling != NULL) {
        if ((p->sibling)->n < min) {
            min = (p->sibling)->n;
            t = p;
            x = p->sibling;
        }
        p = p->sibling;
    }
    if (t == NULL && x->sibling == NULL)
        H1 = NULL;
    else if (t == NULL)
        H1 = x->sibling;
    else if (t->sibling == NULL)
        t = NULL;
    else
        t->sibling = x->sibling;
    if (x->child != NULL) {
        REVERT_LIST(x->child);
        (x->child)->sibling = NULL;
    }
    H = bin_HEAP_UNION(H1, Hr);
    return x;
}

int REVERT_LIST(struct node* y) {
    if (y->sibling != NULL) {
        REVERT_LIST(y->sibling);
        (y->sibling)->sibling = y;
    }
    else {
        Hr = y;
    }
}
/**********binomial heap****************/
```

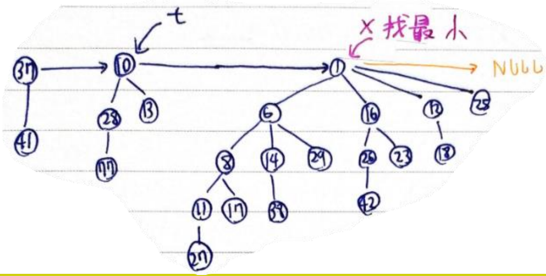
此 while 迴圈
找出 min 的值
將 min 的節點的上一個存入 t，
min 的節點為 x

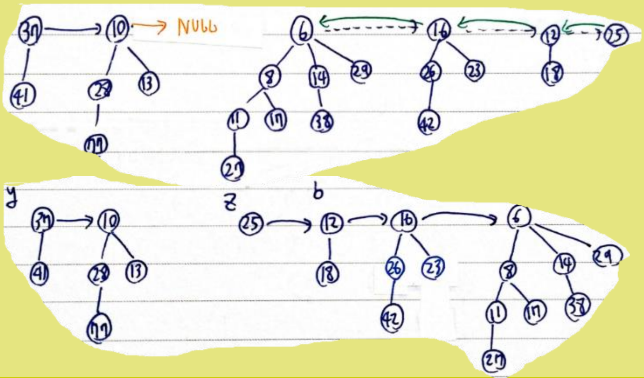
讓原先的 heap 在移除某個擁有 minimum 的子 heap 後，能夠還是維持一個 binomial heap1。

讓有 minimum 的子 heap 的 child 先串起來成為 binomial heap2。

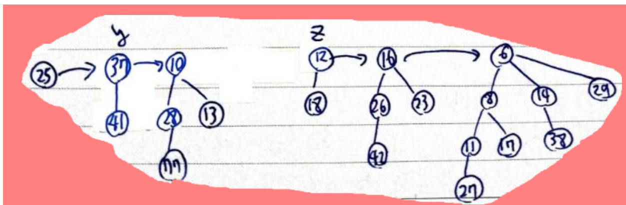
讓 binomial heap1 與 binomial heap2 聯集

將最小的節點**刪掉**，舉例:



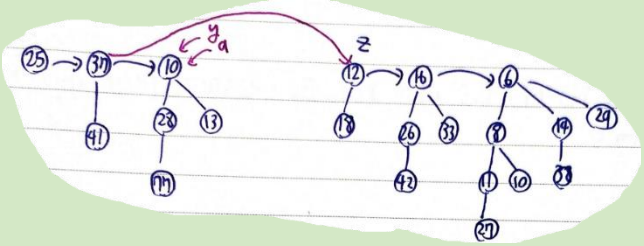


反轉，因為一個 Heap 都是把子 heap 中節點最大的放在最左邊，但先在要 Merge，必須換方向➔REVERT_LIST



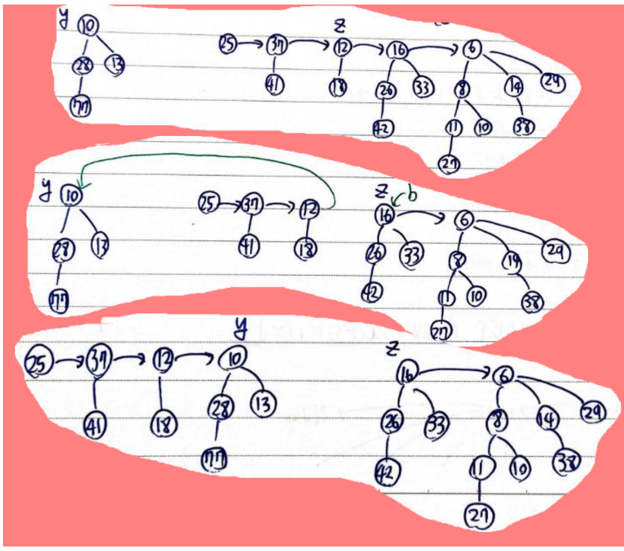

```
else {
  b = z->sibling;
  z->sibling = y;
  z = b;
```

當 y 的 degree 大於 z 的 degree，將 z 往後移一個，原先的 z 連到 y 前面。

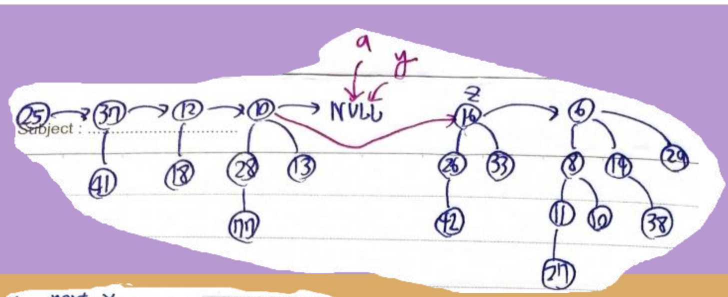



```
else if (y->degree == z->degree) {
  a = y->sibling;
  y->sibling = z;
  y = a;
```

當 y 的 degree 跟 z 的 degree 相等，就將 y 往後移一個，原先的 y 連到 z 前面

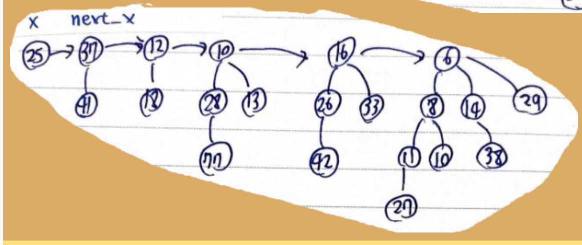



```
else {
  b = z->sibling;
  z->sibling = y;
  z = b;
```




```
if (y->degree < z->degree) {
    y = y->sibling;
```

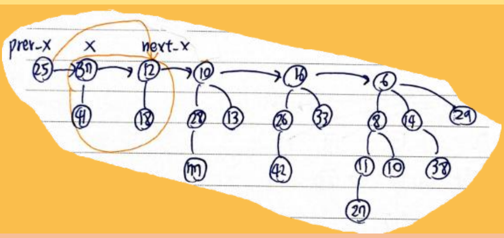
當 y 的 degree 小於 z 的 degree，將 y 連到 z 前面。
遞迴結束



當 x 的 degree 不等於兄弟的 degree，(此例子符合)或是，x 的 degree 等於兄弟的兄弟的 degree

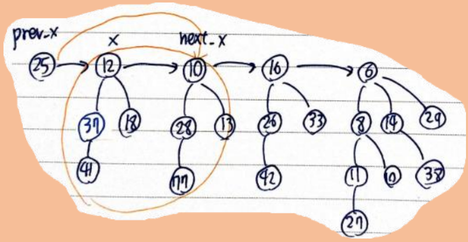
➔所有指標往前移一個

```
if ((x->degree != next_x->degree) || ((next_x->sibling != NULL) && (next_x->sibling)->degree == x->degree)) {
    //如果x的degree不等於x兄弟的degree，或是x兄弟的兄弟degree等於x的degree。
    prev_x = x;
    x = next_x;
}
```

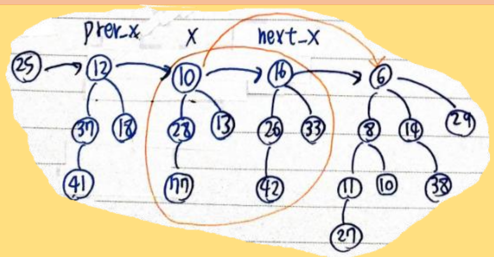



```
else {
    if (prev_x == NULL){
        H = next_x;
    }
    else{
        prev_x->sibling = next_x;
    }
    bin_LINK(x, next_x);
    x = next_x;
}
```

x 的 degree 等於 x 兄弟的 degree 但等於 x 兄弟的兄弟的 degree，x 的值大於兄弟的值




```
else {
    if (prev_x == NULL){
        H = next_x;
    }
    else{
        prev_x->sibling = next_x;
    }
    bin_LINK(x, next_x);
    x = next_x;
}
```



當 x 的 degree 不等於兄弟的 degree，或是，x 的 degree 等於兄弟的兄弟的 degree(此例子符合)

➔所有指標往前移一個

```
if ((x->degree != next_x->degree) || ((next_x->sibling != NULL) && (next_x->sibling)->degree == x->degree)) {
    //如果x的degree不等於x兄弟的degree，或是x兄弟的兄弟degree等於x的degree。
    prev_x = x;
    x = next_x;
}
```

```c
/***********leftist heap****************/
typedef struct leftist Tree;

struct leftist{
    Tree * leftChild;
    int data;
    Tree * rightChild;
    int shortest;
};

typedef int bool;
enum { false, true };
void SWAP(Tree *a,Tree *b,Tree* temp){
    *temp = *a;
    *a = *b;
    *b = *temp;
}
Tree* minUnion(Tree *a,Tree *b){
    /*recursivelu combine two nonempty min leftist trees */
    Tree* temp =(Tree *) malloc(sizeof(Tree));
    /* set a to be the tree with smaller root*/

    if(a->data > b->data){
        SWAP(a,b,temp); //右子樹的數目比較多，先調整
    }
    if(a->rightChild==NULL){            //如果沒有又子樹的話，就將b直接當作a的左子樹
        a->rightChild = b;
    }
    else{

        minUnion(a -> rightChild,b); //遞迴呼叫
    }
    //滿足leftchild的特性
    if(a->leftChild==NULL){            //沒有左小孩  ，將右小孩作為左小孩。
        a->leftChild = a->rightChild;
        a->rightChild = NULL;
    }
    else if(a->leftChild->shortest < a->rightChild->shortest){ //左小孩的數量比右小孩小
        SWAP(a->leftChild,a->rightChild,temp);
    }
    a->shortest = (!a->rightChild) ? 1: a->rightChild->shortest +1;
    return a;
}
void minMeld(Tree *a,Tree *b){
    if(a==NULL) *a = *b;
    else if(b!=NULL) minUnion(a,b);
    b = NULL;
}
void delete_min(Tree *a){
    if(a!=NULL){
        Tree * b  = (Tree *) malloc(sizeof(Tree));
        if(a->rightChild!=NULL){
            *b = *(a->rightChild);
            *a = *(a->leftChild);
            minMeld(a,b);
        }
        else{
            *a = *(a->leftChild);
        }
    }
}
/******************leftist heap******************************/
```

```c
int main() {
    int i, n, m, l,op;
    struct node* p;
    struct node* np;
    char ch;
    srand(time(NULL));


    printf("\nEnter the number of elements:");
    scanf("%d",&n);


    printf("\nEnter the number of operation:");
    scanf("%d",&op);

    int op_array[op];
    for (i = 0; i< op; i++){
        op_array[i] = rand()%2;
        //printf("%d",op_array[i]);
    }

    int start1 = clock();
    for (i = 1; i <= n; i++) {
        m = rand()%n;
        np = CREATE_NODE(m);
        H = bin_HEAP_INSERT(H, np);
    }
    for (i = 0;i< op;i++){
        switch (op_array[i]){
            case 0:
                m = rand()%n;
                p = CREATE_NODE(m);
                H = bin_HEAP_INSERT(H, p);
                break;
            case 1:
                p = bin_HEAP_EXTRACT_MIN(H);
                break;
        }
    }
    double elapsedTime1 = (double) (clock() -start1)/CLOCKS_PER_SEC;
    printf("elapsedTime (binomial heap): %.8f\n",elapsedTime1);

    Tree *a = (Tree *) malloc(sizeof(Tree));
    Tree *b;
    a->shortest = 1;
    a->leftChild = NULL;
    a->rightChild = NULL;

    start1 = clock();

    for (i = 1; i <= n; i++) {
        b  = (Tree *) malloc(sizeof(Tree));
        b->shortest = 1;
        b->leftChild = NULL;
        b->rightChild = NULL;
        int data = (rand()%n)+1;
        b->data = data;
        a =minUnion(a,b);
    }
    for (i = 0;i< op;i++){
        switch (op_array[i]){
            case 0:
                b  = (Tree *) malloc(sizeof(Tree));
                b->shortest = 1;
                b->leftChild = NULL;
                b->rightChild = NULL;

                int data = (rand()%n)+1;
                b->data = data;
                a =minUnion(a,b);
                break;
            case 1:
                delete_min(a);
                break;
        }
    }
    elapsedTime1 = (double) (clock() -start1)/CLOCKS_PER_SEC;
    printf("elapsedTime (leftist heap): %.20f\n",elapsedTime1);
}
```

一個 Binomial Heap 有多少個 root?

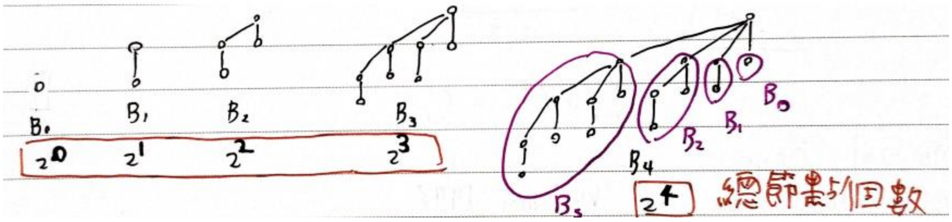
數學推導:

一個 Binomial Heap 最少有一個 Binomial tree，最多有 k 個 Binomial tree， 。

B_{k-1} B_{k-2} B_{k-3} ...... B_1 B_1 有 K 個 binomial tree。且 B_{k-1} 不為 0。

因為 binomial tree 的特性為 ：

Head node 的 degree 為 k 則 binomial tree 的節點個數為 2^k。



所以整個 Binomial Heap 的總個數 N

$2^{k-1} <= N <= 2^{k-1} + 2^{k-2} + ... + 2^2 + 2^1 + 2^0 = 2^k - 1$

$N+1 <= 2^k <= 2N$

$Log_2(N+1) <= Log_2(2^k) <= Log_2(2N)$

$Log_2(N+1) <= K <= 1 + Log_2(N)$

所以有多少顆 binomial tree➔ K 相當於 $Log_2(N)$

Delete

Binomial Heap 是一堆 binomial tree 的集合。

如果最大的 degree 為 3。則 heap 的排列組合方式有。

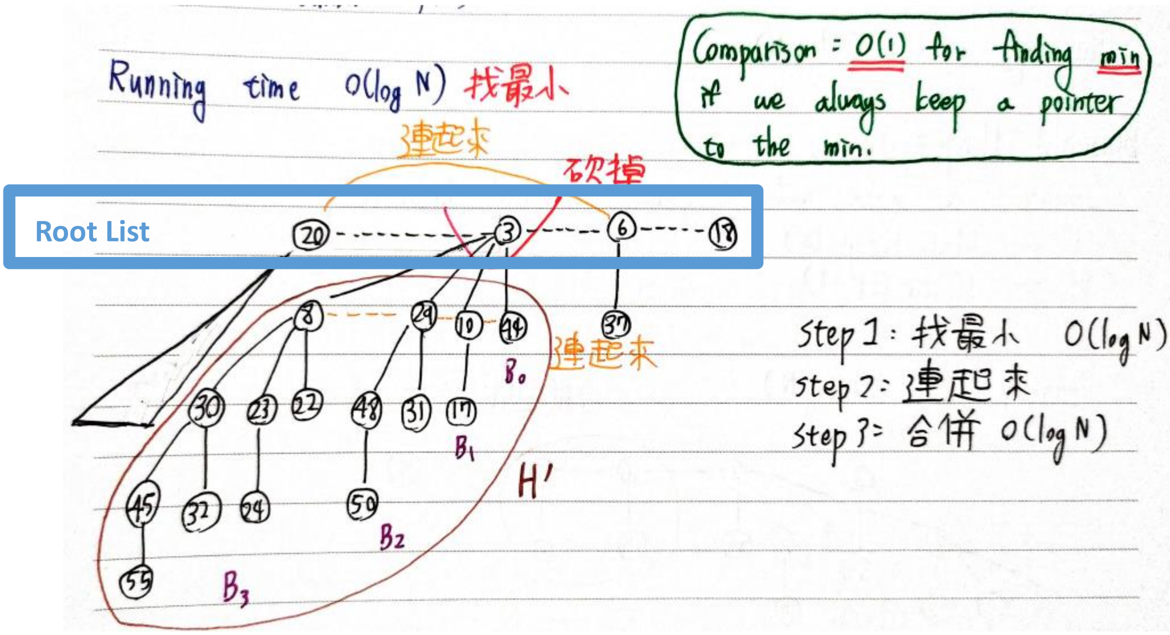
(0,4) or (1,3,4) or (0,1,2,4) or (0,1,2,3,4)。

其中的數字是 binomial tree 的 minimum 的 degree。

如果要 delete minimum 則是在 binomial heap 的 root list 中找最小值

如下圖:在 root list 找到最小值 3 並且把 3 砍掉，silbling 相連，變成新的 Heap H

3 的 child 連成一個新的 Heap H' 再將 Heap H 與 Heap H' union。



合併的時間複雜度

把兩個 Binomial Heap 的 root list 逛過一次(MERGR)再做（ＵＮＩＯＮ）如之前的圖示範。而這些都是在採訪 root list 的每個節點所以時間複雜度是

Ｏ（Log_2(N)＋Log_2(N)）

而刪除動作要先找最小值＝＞Ｏ（Log_2(N)）

刪掉後再ＵＮＩＯＮ兩個 binomial tree＝＞Ｏ（Log_2(N)＋Log_2(N)）

相加後時間複雜度仍然是Ｏ（Log_2(N)）。

插入的時間複雜度

插入相當於建立一個只有一個節點的 Binomial Heap。而在將這個 Binomial Heap 與原先的 Binomial Heap　ＵＮＩＯＮ　時間複雜度為Ｏ（Log_2(N)＋Log_2(N)）

但因為每次插入的只有一個節點　所以相當於只有原先的 Binomial Heap 的 root list 需要採訪，因此時間複雜度為Ｏ（Log_2(N)），但因為插入每次都只差到第一個所以有兩種 case：會進位與不會進位。

插入一個新的節點到 binomial heap H

H 原本的 binomial heap represent:

H : …………0 插入後不進位　，只需一個步驟 ➔出現的機率 1/2

H : …………01 插入後進位一次　，只需兩個步驟➔出現的機率(1/2)*(1/2)

H : …………011 插入後進位兩次　，只需三個步驟➔出現的機率(1/2)*(1/2))*(1/2)

H : …………0111 插入後進位三次　，只需四個步驟➔出現的機率(1/2)*(1/2) *(1/2) *(1/2)

因此插入 N 個節點 每個節點插入所需的步驟➔ 1*(1/2)+2*(1/4)+ 3*(1/8)+ 4*(1/16)+….

$\sum_{n=1}^{N} N/2$^n<=2N 。因次每個插入動作所需的步驟是 2N/N ➔ 1

因此可以說插入動作的時間複雜度為 O(1)

因此 Binomial Heap 操作的時間複雜度為(½ * O(1) + ½ * O(log_2(N)))，也就是 O(log_2(N))

Leftist Heap from 作業 20

做刪除動作:

statement	frequency	Total steps
Tree * b = (Tree *) malloc(sizeof(Tree));		
int _id = a->id;	1(因為有 assign)	1
if(a->rightChild!=NULL){	1(要刪除的有兩個小孩) 因為是左傾，所以如果有右小孩一定有左小孩	1
*b = *(a->rightChild);	1	1
*a = *(a->leftChild);	1	1
minMeld(a,b);	1	@@
}		
else{	1(要刪除的節點有可能只有一個小孩或沒有小孩)	1 這個 else 都是 O(1)所以影響不大
*a = *(a->leftChild);	1	1
}		
return _id;		

插入的話 就是先做合併動作

做合併動作 minMeld

statement	frquency	Total steps
if(a==NULL)	1	1
*a = *b;	1 如果只有一個節點直接給 assign 給 a	1
else if(b!=NULL)	1 如果 a 跟 b 都非空	1
minUnion(a,b);	1 就呼叫另一個函數	@@
b = NULL;	1	1

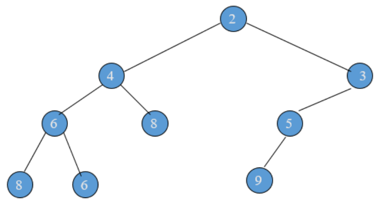
statement	frequency	totalsteps
Tree* temp =(Tree *) malloc(sizeof(Tree));	1	1
if(a->data.key > b->data.key){	1 //右子樹的的大小比較大，先調整	1
SWAP(a,b,temp);	1	3
}		
if(a->rightChild==NULL){	1 //如果沒有右子樹的話，就將 b 直接當作 a 的左子樹	1
a->rightChild = b;	1	1
}		
else{	有右子樹會做 O(logn)為 rightmost path 的長度	
minUnion(a -> rightChild,b);	遞迴呼叫	
}		
if(a->leftChild==NULL){	1 如果沒有左小孩，則左右小孩交換	1
a->leftChild = a->rightChild;	1	1
a->rightChild = NULL;	1	1
}		
else if(a->leftChild->shortest < a->rightChild->shortest){	1	1
SWAP(a->leftChild,a->rightChild,temp);	1	3
}		
a->shortest = (!a->rightChild) ? 1: a->rightChild->shortest +1;	1	1
return a;		

灰色的 statement 是看狀況才會執行，但除了遞迴呼叫外，其餘的時間複雜度都為 O(1)，倘落有進到遞迴呼叫，會進行 O(log n) 次 ，n 為 number of nodes in the a leftist tree.

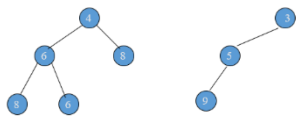
每次 minUnion 的時間複雜度為因為其他都是 O(1)，所以可以說 Union 複雜度是 O(log n)。

因此刪除動作的複雜度 O (log n)。(圖解)

刪除最小的節點

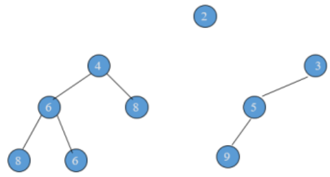


再將兩個子樹合併。

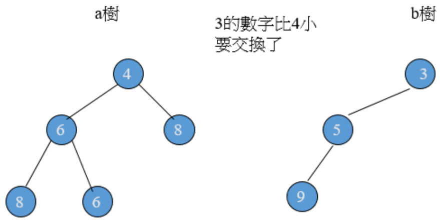


而合併相當於採訪最右邊的節點到root的長度。

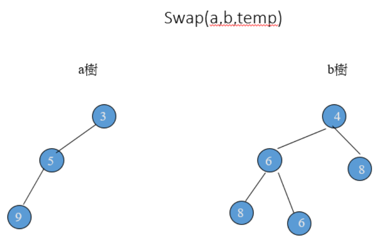
刪除最小的節點

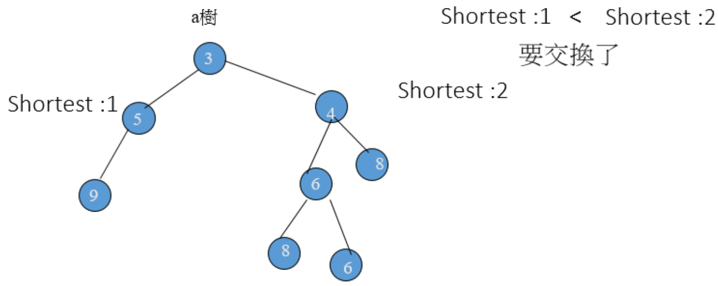


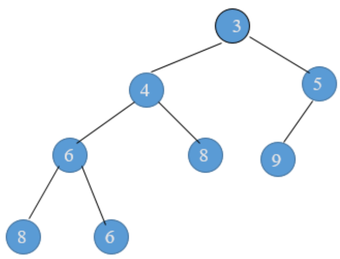
相當於把左子樹與右各自分開。形成兩個leftist tree



最右節點到 root 的長度為 log(n)。



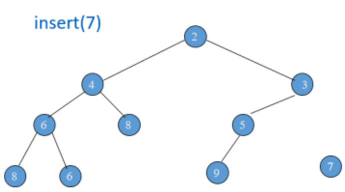




要算左右小孩的 shortest 所以需要 log(n) ➔ 樹的高度的時間。

而插入動作其實就是合併動作 O(log n)。但差別差在要先建立一個 node。所以時間複雜度為 O(1)

Insert Operation



插入節點相當於先建立一個單一節點的 leftist tree，再把新的tree跟原本的tree合併.

Leftist heap 插入的頻率與刪除的頻率各一半，(½ * O(log n) + ½ * O(log n))

Binomial Heap 操作的時間複雜度為(½ * O(1) + ½ * O($\log_2$(N))

所以可以得到的結論是 Binomial Heap 的操作用時較 Leftist heap 少

實驗結果:

Binomial heap 的用時都小於 Leftist heap 符合預期

```
選取 E:\資料結構\資料結構作業三\f74109016_hw21.exe

Enter the number of elements:100

Enter the number of operation:5000
elapsedTime (binomial heap): 379.300000us  每個operation time 0.075860us
elapsedTime (leftist heap): 803.500000us  每個operation time 0.160700us
```

```
E:\資料結構\資料結構作業三\f74109016_hw21.exe

Enter the number of elements:500

Enter the number of operation:5000
elapsedTime (binomial heap): 365.800000us  每個operation time 0.073160us
elapsedTime (leftist heap): 1342.300000us  每個operation time 0.268460us
```

```
E:\資料結構\資料結構作業三\f74109016_hw21.exe

Enter the number of elements:1000

Enter the number of operation:5000
elapsedTime (binomial heap): 411.300000us  每個operation time 0.082260us
elapsedTime (leftist heap): 1869.800000us  每個operation time 0.373960us
```

```
E:\資料結構\資料結構作業三\f74109016_hw21.exe

Enter the number of elements:2000

Enter the number of operation:5000
elapsedTime (binomial heap): 1013.800000us  每個operation time 0.202760us
elapsedTime (leftist heap): 3333.600000us  每個operation time 0.666720us
```

```
E:\資料結構\資料結構作業三\f74109016_hw21.exe

Enter the number of elements:3000

Enter the number of operation:5000
elapsedTime (binomial heap): 664.700000us  每個operation time 0.132940us
elapsedTime (leftist heap): 3648.200000us  每個operation time 0.729640us
```

```
E:\資料結構\資料結構作業三\f74109016_hw21.exe

Enter the number of elements:4000

Enter the number of operation:5000
elapsedTime (binomial heap): 768.600000us  每個operation time 0.153720us
elapsedTime (leftist heap): 4237.600000us  每個operation time 0.847520us
```

```
E:\資料結構\資料結構作業三\f74109016_hw21.exe

Enter the number of elements:5000

Enter the number of operation:5000
elapsedTime (binomial heap): 880.300000us  每個operation time 0.176060us
elapsedTime (leftist heap): 4929.900000us  每個operation time 0.985980us
```

		微秒			
element數量	operation 數量	binomial heap	leftist heap	binomial heap每個operation time	leftist heap每個operation time
100	5000	379.3	803.5	0.07586	0.1607
500	5000	365.8	1342.3	0.07316	0.26846
1000	5000	411.3	1869.8	0.08226	0.37396
2000	5000	1013.8	3333.6	0.20276	0.66672
3000	5000	664.7	3648.2	0.13294	0.72964
4000	5000	768.6	4237.6	0.15372	0.84752
5000	5000	880.3	4929.9	0.17606	0.98598