**tags: 資料結構**

# HW20

本題需要比較左偏樹以及堆的查詢元素、查詢堆頂元素、刪除堆頂元素，因前一項功課已經介紹過小頂堆，故此不再重複敘述。

## Leftist tree

左偏樹是堆推廣而來的資料結構，因此可以實現堆的所有操作，且複雜度也相同，比較不一樣的是左偏樹支持在 $O(\log_2 n)$ 內實現合併兩個堆的操作，這是普通堆做不到的，普通堆需要將整個堆拆散重新插入，因此需要 $O(N)$ 才能實現。

首先定義以下名詞用來解釋左偏堆性質。

- $val$，代表該節點的權值
- $npl$， `Null path length` 又稱零路徑長，代表該節點到沒有兩個子節點(外節點)的節點的長度
- $L$，指向左節點
- $R$，指向右節點

## Nature

定義完名詞後可以知道左偏樹會有以下幾個重要的性質。

- 堆性質
  - $val(x) < val(L(x))$ 且 $val(x) < val(R(x))$
- 左偏性質 1
  - 對於任意 $x$ 成立 $dis(R(x)) \leq dis(L(x))$
- 左偏性質 1
  - 對於任意 $x$ 成立 $dis(x) = dis(R(x)) + 1$

對於左偏性質可以很好的知道他維護了一棵樹的左偏型態，而對於每一棵子樹，則盡量使右兒子離根節點近。而有關於 $dis$ 的部分我們可以由下列證明推導出其值最大為 $log_2(n+1) - 1$。

對於一棵節點數為 $n$ 且最大距離為 $k$ 的左偏樹，根據二叉樹的基本性質可知道至少會有 $2^{k+1}-1$ 個節點，因此可以得到下列公式。

$$n \geq 2^{k+1}-1$$

$$\Rightarrow n+1 \geq 2^{k+1}$$

$$\Rightarrow \log_2 n+1 \geq k+1$$

$$\Rightarrow \log_2(n+1)-1 \geq k$$

## Merge

假設現在需要合併 $x$、$y$ 兩個堆，為了讓時間複雜度盡可能的小，所以我們需要讓樹的 $level$ 也盡可能的小。根據左偏樹的性質我們已經維護右子的 $dis$ 盡可能小了，所以我們可以將 $merge(x,y)$ 的操作變成 $merge(R(x),y)$，需要注意的是我們同時還得維護 $val(x)<val(y)$ 的性質。

合併完成後由於右節點可能會發生變化導致**左偏性質 1**、**2**不成立，因此需要對右節點及其根節點更新，而由之前的證明可知節點為 $n$ 的左偏樹，其最大 $dis$ 為 $\log_2(n+1)-1$，那麼合併的最大時間度及為下列。

$$i \in tree(x), j \in tree(y)$$

$$O(max(dis(i))+max(dis(j)))$$

$$\Rightarrow O(2\log_2(n+1)-2)$$

$$\Rightarrow O(\log_2 n)$$

以下為 pseudo code

```
Node* _merge(Node* x, Node* y)
{
    if (x->left == NULL)
        x->left = y;
    else {
        x->right = merge(x->right, y);
        if (x->left->npl < x->right->npl)
            swap(x->left, x->right);
        x->npl = x->right->npl + 1;
    }
    return x;
}

Node* merge(Node* x, Node* y)
{
    if (x == NULL)
        return y;
    if (y == NULL)
        return x;
    return (x->val < y->val ? _merge(x, y) : _merge(y, x));
}
```

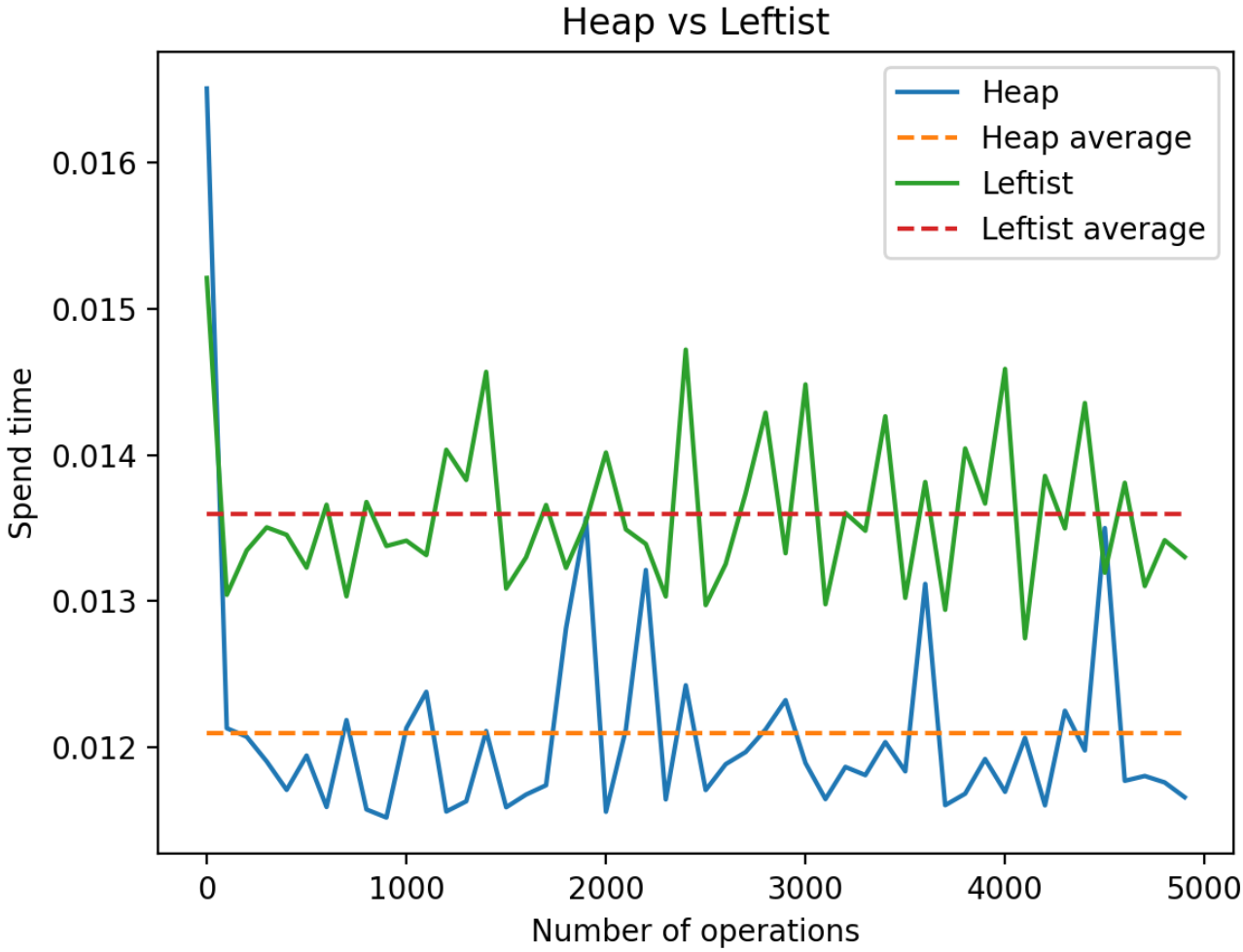
## Insert and Destroy

完成 merge 後基本上就通通完成了，剩下 insert 及 destroy 操作都是 merge 的基本延伸，insert 是將要插入的點視為一棵新樹與原樹做 $merge$，而 destroy 則是將 $R(x)$ 及 $L(x)$ 做 merge，以下為 pseudo code。

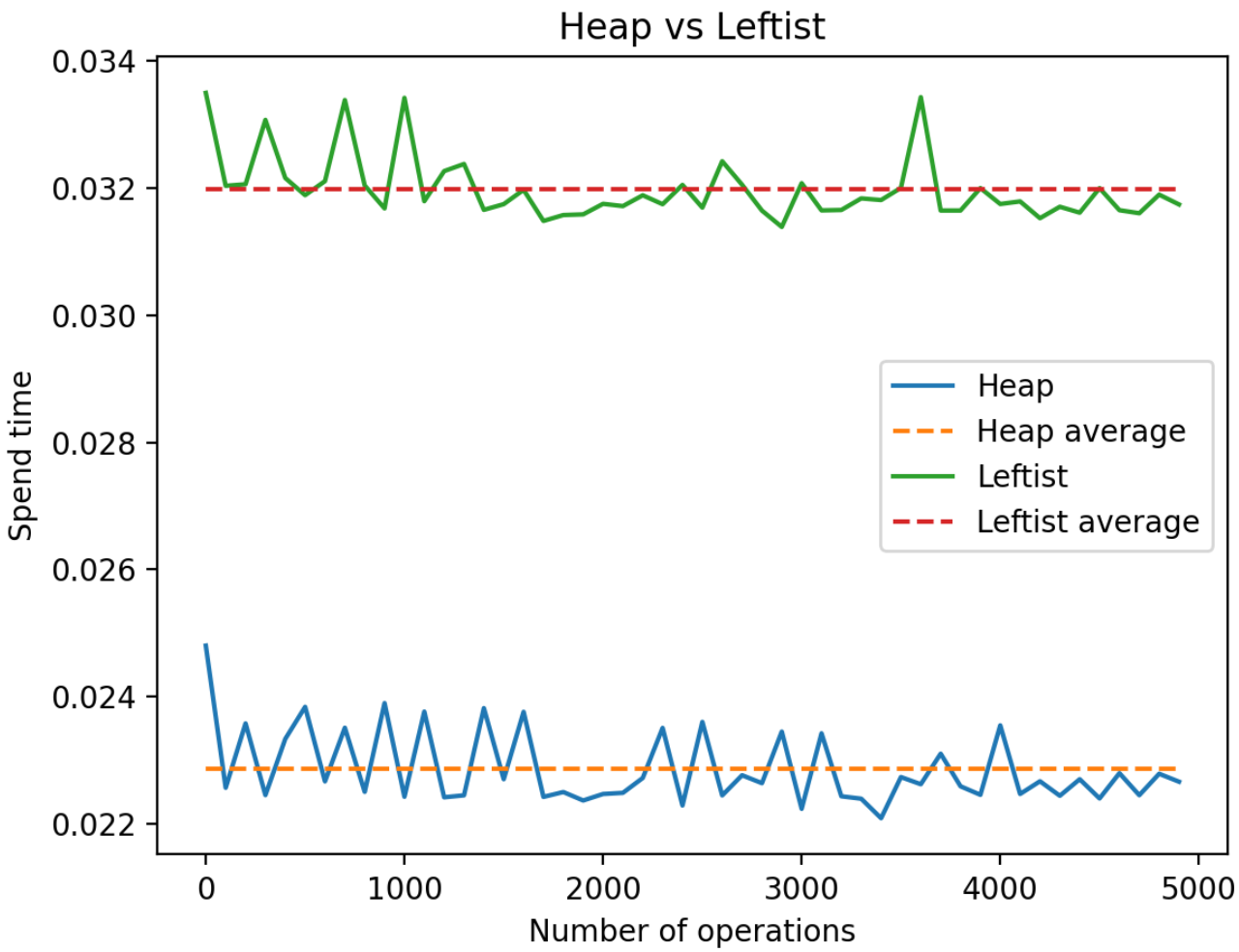
```
Node* insert(Node* h, int x)
{
    Node* tmp = init(x);
    h = merge(h, tmp);
    return h;
}

Node* destroy(Node* h)
{
    return merge(h->left, h->right);
}
```

# Compare

接下來是 Heap 與 Leftist 的比較，這邊我寫了一個腳本(請見此處(https://github.com/arasHi87/NCKU_HOMEWORK/blob/master/Data%20Structure/hw20/measure.py))用來產生測資以及比對兩者的差距，首先我產生了 $10^4$ 個元素用來初始化，接著兩者分別都做了 1 ~ 5000 次操作，下圖是紀錄表。




可以看到兩者基本上沒有差別，主要是差在了 Leftist tree 用到了較多指標操作，因此會有常數差距，接著我將測資拉到 $10^5$ 做了下列觀察，可以看到差距有稍微拉開了，因此可以知道指標對比陣列操作還是會不可避免的產生常數。




根據上述的觀察，對於兩者的優劣比較下我會認為如果只是要單純做最值相關操作的話使用 Heap 還是會較好，但如果有需要合併堆的情況下則是 Leftist tree 會更勝一籌。