HW4 Report

Q1:

編譯：

```
g++ main.cpp -o main
```

執行：

```
./main
```

Q2:

b tree 與 b+ tree 有許多差別，下述兩項最為重要：

- data pointer
  - b tree 中的 internal node 與 leaf node 都有 data pointer
  - b+ tree 中僅 leaf node 有 data pointer

- leaf node
  - b tree 中的 leaf node 並沒有 linked
  - b+ tree 中的 leaf node 有 linked

因為 data pointer 都在 leaf node 的關係，使得 b+ tree 在 insertion、deletion 的過程中都比 b tree 更快且有效率。

此外，因為 b+ tree 的 leaf node 有 linked 在一起，透過 linear search 就可以 access 所有的 data pointer；然而，在 b tree 中需要 traverse 整顆 tree。

Q3:

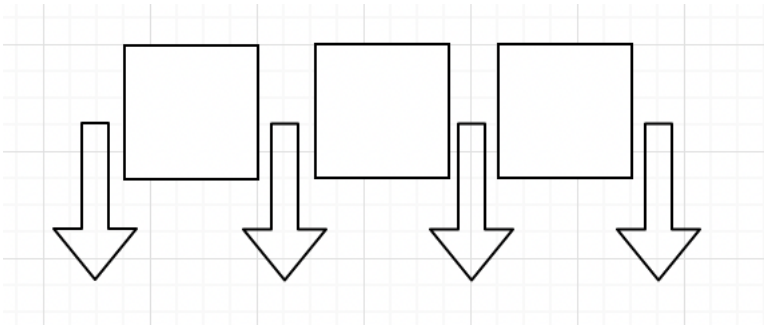
```
class Node
{
    public:
        int* keys;
        int num_keys;
        Node** child_nodes;
        Node* parent_node;
        bool is_leaf;

        Node(int max_key_num, int max_child_num) {
            keys = new int[max_key_num];
            num_keys = 0;
            child_nodes = new Node*[max_child_num];
        }
};
```

本次作業中，一個 Node 包含了 5 個 data member：

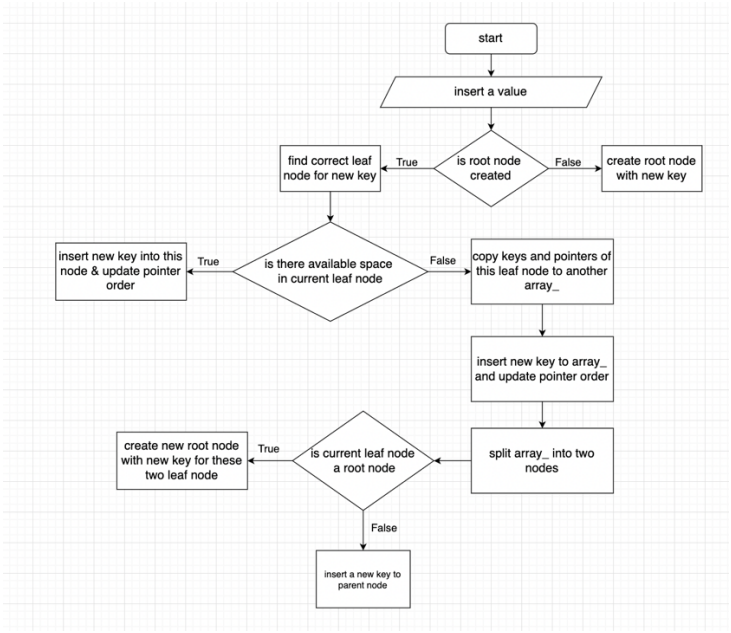
- keys: 指向 integer 的 pointer。用來存放這個 node 中所有的 key。
- num_keys: 一個 integer。記錄目前 node 中有多少 key。
- child_nodes: 指向「node 的 pointer」的 pointer。用來存放這個 node 中所有「指向 child node 的 pointer」。
- parent_node: 指向 node 的 pointer。用來存放指向 parent node 的 pointer。
- is_leaf: boolean 值。用來表示這一個 node 是不是 leaf node。

在 b+ tree 中，key 與 pointer 的位置如圖所示：



因此，透過 keys (dynamic allocated array) 存放 n 個 key，透過 child_nodes (dynamic allocated array) 存放 n+1 個 pointer。

Q4:



實作描述：

在 insert 新的 key 進入 b+ tree 時，先檢查 root node 是否已經建立，如果還沒建立，則先建立一個 root node。

```
// root node is not created
if(root == NULL) {

    if(is_debug) {
        cout<<"<INFO>: root node is not created "<<value<<endl;
    }

    root = new Node(max_key_num, max_child_num);
    root->keys[0] = value;
    root->num_keys = 1;
    root->is_leaf = true;
}
```

如果已經建立了 root node，則必須尋找這個 key 正確的 leaf node。

```
while(current_node->is_leaf == false) {
    for(int i=0; i<current_node->num_keys; i++) {
        if(value < current_node->keys[i]) {
            current_node_parent = current_node;
            current_node = current_node->child_nodes[i];
            break;
        }

        if(i == current_node->num_keys-1) {
            current_node_parent = current_node;
            current_node = current_node->child_nodes[i+1];
            break;
        }
    }
}
```

接著，判斷這一個 leaf node 有沒有空間再插入一個 key。如果有空間，則插入新的 key 到正確的位置，並更新 pointer 順序。如果沒有，則必須進行 leaf node split。

實作上，先將這個 leaf node 中所有的 key 存到一個 array 中。

```
// duplicate all keys in current node
int duplicate_keys[max_key_num + 1];
for(int i=0; i<max_key_num; i++) {
    duplicate_keys[i] = current_node->keys[i];
}
```

並找到新的 key 在這一個 array 中的位置，並插入進去。

```
// find position of new value in duplicate keys array
int idx = 0;
while(idx < max_key_num) {
    if(value < duplicate_keys[idx]) {
        break;
    }
    idx++;
}
```

```
// insert new value into duplicate keys array
for(int j=(max_key_num+1)-1; j>idx; j--) {
    duplicate_keys[j] = duplicate_keys[j-1];
}
duplicate_keys[idx] = value;
```

建立兩個 node : first_leaf_node (原來的 leaf node) 與 second_leaf_node。並設定好這兩個 node 應該擁有的 key 的數目。

```
// store keys in duplicate keys array into first leaf node and second leaf node
Node* first_leaf_node = current_node;
first_leaf_node->num_keys = int((max_key_num+1)/2);

Node* second_leaf_node = new Node(max_key_num, max_child_num);
second_leaf_node->num_keys = (max_key_num+1) - int((max_key_num+1)/2);
second_leaf_node->is_leaf = true;
```

再將 array 中的 key 依照兩個 node 應該擁有的 key 的數目，將 key 分配給這兩個 node。

```
for(int i=0; i<first_leaf_node->num_keys; i++) {
    first_leaf_node->keys[i] = duplicate_keys[i];
}

for(int i=0; i<second_leaf_node->num_keys; i++) {
    second_leaf_node->keys[i] = duplicate_keys[i+first_leaf_node->num_keys];
}
```

再更新兩個 node 的 pointer 順序，使得 leaf node 中最後一個 pointer 指向下一個 leaf node。

```
first_leaf_node->child_nodes[first_leaf_node->num_keys] = second_leaf_node;
second_leaf_node->child_nodes[second_leaf_node->num_keys] = first_leaf_node->child_nodes[max_key_num];
first_leaf_node->child_nodes[max_key_num] = NULL;
```

最後，必須為新建立的 node (second_leaf_node) 建立一個 parent node。如果原本的 leaf_node (first_leaf_node) 是 root node 時，則在建立一個新的 root node；如果不是，則以 recursive 的方式在 internal node 中插入一個新的 key (second_leaf_node 中的第一個 key)。

```
// build parent for second leaf node
if(first_leaf_node == root) {
    Node* new_root = new Node(max_key_num, max_child_num);
    new_root->keys[0] = second_leaf_node->keys[0];
    new_root->num_keys = 1;
    new_root->is_leaf = false;
    new_root->child_nodes[0] = first_leaf_node;
    new_root->child_nodes[1] = second_leaf_node;
    root = new_root;

    first_leaf_node->parent_node = new_root;
    second_leaf_node->parent_node = new_root;
} else {
    insertInternalValue(second_leaf_node->keys[0], current_node_parent, second_leaf_node);
}
```

Q5:

範例一：

```
johnny@johnny-VirtualBox:~/program$ ./main < 0.in
()

(50)
  (10)
  (50 90)

(50)
(10)
QAQ

(50)
(50 90)
Found

(30)
  (20)
    (10)
    (20)
  (50)
    (30 40)
    (50 90)
```

範例二：

```
johnny@johnny-VirtualBox:~/program$ ./main < 5.in
(60)
  (30 45)
    (10 15 20 25)
    (30 35 40)
    (45 50 55)
  (70 80)
    (60 65)
    (70 75)
    (80 85 90)

Access Failed

55 60 65 70

55 60 65 70 75 80 85 90
N is too large
```