• Priority Queues (search key + priority value)

▸ Heap (實現的其中一個方法)

• 用選擇排序 (先放入,再排) 用二元樹 (邊放邊排), 效率最好：balance tree

Insert: O(1), Delete: O(n) Insert: O(n), delete: O(n)

插入排序 Insert: O(n), Delete: O(1)

min-heap 找小

max-heap 找大

• Void ReheapDown (int, int);

√oid ReheapUp (int, int);

Le the rightmost node at the last level (bottom) of the tree

[由下而上放入,由機會在中間就停止]

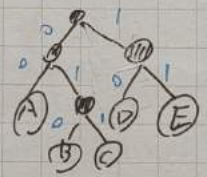
▲(MAX) Insert: O(log n)

▸基本性質:

• complete binary tree

• the value stored at a node is greater (smaller) or equal to the values stored at the children (heap property)

▸應用：壓縮影像碼



A: 00 D: 10

B: 010 E: 11

C: 011

Probabilities → weight

```
• Void heapInsert (HeapItemType & newItem);
{ if (size >= MAX_HEAP)
      throw HeapException("Heap full");
  Items[size] = newItem;
  int place = size;
  int parent = (place-1)/2;
  while ((parent >= 0) && (items[place] > items[parent]))   MAX heap
  { HeapItemType temp = items[parent];
    items[parent] = items[place];
    items[place] = temp;
    place = parent;
    parent = (place-1)/2; } ++size; }
```

```
• void heapRebuild(int root); [use recursive]
// Converts the semi-heap rooted at index root into a heap
   這樹根不對
{ int child = 2 * root + 1;
  if (child < size)
  { int rightChild = child + 1;
    if ((rightChild < size) && (items[rightChild] > items[child]))
        child = rightChild;
    if (items[root] < items[child])   MAX heap
    { HeapItemType temp = items[root];
      items[root] = items[child];
      items[child] = temp;
      heapRebuild(child);
    }
  }
```

```
• void heapDelete (HeapItemType& rootItem);
節點由1點,將 bottom rightmost 移上去空位,
變成 semi-heap
  { if heapIsEmpty()
      throw HeapException("Heap Empty");
    rootItem = items[0];
    --size;
    if !heapIsEmpty()
    { item[0] = item[size];
      // move the last item (bottom) in the
      heap to the root
      heapRebuild(0);
        // trickles it down to its
           proper position.
    }
  }
```

Heap Sort: Pseudocode

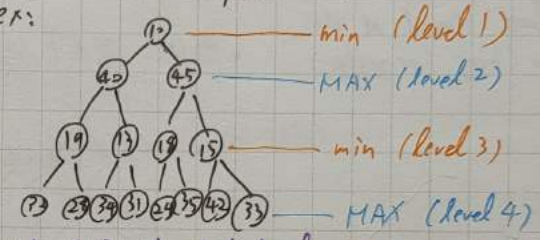
```
void heapSort(int size){
    int index;
    // convert array[0...size-1] into a heap.
    for (index=(size-1)/2; index>=0; index--)
        ReheapDown(index, size);      <=> heapRebuild(index)
    // Sort the array.
    for (index = size-1; index>=1; index--){
        Swap(items[0],items[index]);
        ReheapDown(index, index);     <=> heapRebuild(0)
    }
}
```

Variations of Heap

- Double-ended Priority Queues (DEPQ)
  - Min-max Heap
  - Double-ended Heap (DEAP)
- Forest (union) of Heaps
  - Binomial Heap
  - Fibonacci Heap

- Min-max Heap (complete Tree)

ex:



Insert: ① Decide which level → min or MAX
② check whether to swap with its parent
↳ No: ReheapUp from the current node
Yes: ReheapUp from its parent

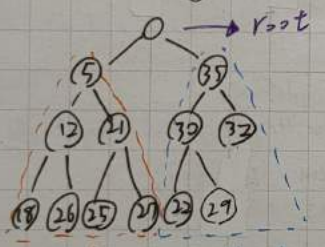
→ max-heap + min-heap + max-heap

→ each node in max-heap has its parent in min-heap

Min-max heap = 1 min-heap + 2 max-heaps

- Double-ended Heap (DEAP) (complete tree)
  - Double-ended Priority Queue (DEPQ)

ex:



Insert: ① Examine the corresponding nodes: Left < Right
② ReheapUp is necessary (recursion)

→ Two heap:

↳ Pseudo root + min-heap + max-heap

↳ each node in max-heap corresponds to one in min-heap

- 考驗排序方法：當東西大到記憶體放不下
- 應用：External sort (外排序)

eg. quicksort + heapsort

- Merge of priority queues (合併排序)
  - Binomial Heap →
    - root has k children
    - Merged by two binomial trees of order k-1
    - Number of nodes = 2^k
    - Tree height = k+1 → O(log n)
    - C_i^k nodes at level i, for i = 0...k
- Fibonacci Heap

The ADT table, or dictionary

- Uses a search key to identify its items
- Its items are records that contain several pieces of data

- Our table assumes distinct search keys
  - Other tables could allow duplicate search keys
- The traverseTable operation visits table items in a specified order
  - One common order is by sorted search key

	Insertion	Deletion	Retrieval	Traversal
Unsorted array based	O(1)	O(n)	O(n)	O(n)
Unsorted pointer based	O(1)	O(n)	O(n)	O(n)
sorted array based	O(n)	O(n)	O(log n)	O(n)
sorted pointer based	O(n)	O(n)	O(n)	O(n)
Binary search tree	O(log n)	O(log n)	O(log n)	O(n)

↑效率與樹高有關

- Height of a binary search tree of n items
  - Maximum: n
  - Minimum: $\lceil \log_2(n+1) \rceil$
  - Sensitive to the order of insertions and deletions

- Nonlinear implementations
  - Binary search tree

Linear

- easy to understand conceptually
- May be appropriate for small tables or unsorted tables with few deletions

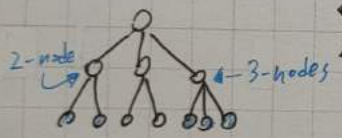
Nonlinear

- Is usually a better choice than a linear implementation
- A balanced binary search tree
  - Increases the efficiency of the table operations

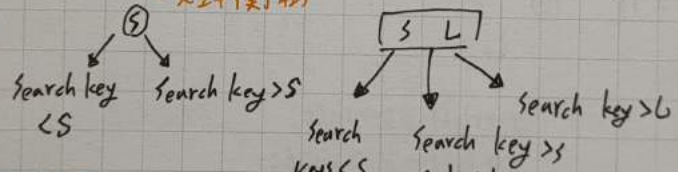
[2-3 tree
[2-3-4 tree

[AVL tree
[Red-black tree

2-3 Tree



- Are general trees, not binary trees
- never taller than a minimum-height binary tree
  ↑完全平衡樹



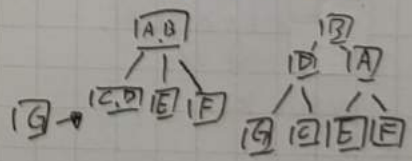
- traverse a 2-3 tree
  - Perform the analogue of an inorder traversal
- Searching 2-3 tree
  - $O(\log_2 n)$ (shortest binary search tree)

刪除 (缺分配)

- Redistribute values
- Merge into a leaf 合併
- Redistribute values and children
- 若是樹根，Delete the root

- 若是分裂根節點
  → 分裂
  → 設立根點
  → 樹高增加

- 插入 2 node 的葉
- 插入 3 node 的葉
  (要分裂 [divide])



2-3-4 Tree

- 遇到 3-nodes 就分開
  (一路上不含有 3-nodes)
  Splits occur only at the path from the root to a leaf (downward)!
  ▲ 不需要 recursion

Delete:

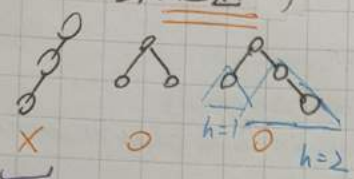
- Transformations occur only at the path from the root to a leaf
  (不需要 recursion)
  遇到1個的，就合併

Summary

- 2-3-4 tree always 平衡
- 2-3-4、2-3、都是用空間換時間
  每個節點有很多key，可能造成空間的浪費

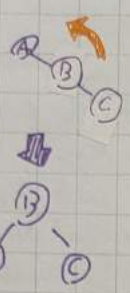
AVL Tree

- tree balanced (左、右差，不超過 1)

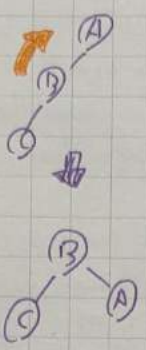


↳ 處理方法 (旋轉) rotate

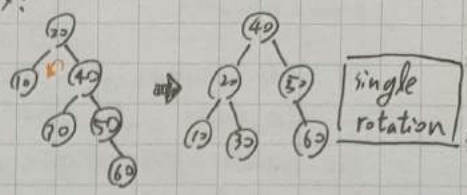
- 左轉

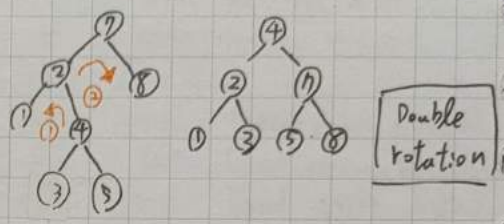


- 右轉



ex:





- 平衡系數 (BF)

BF(a node) = h(left subtree) − h(right subtree)

差必需 ≤ 1 ➡ $-1 \le BF \le 1$

- Height of x→left→left : h+1
  x→left→right : h or h+1

single rotation with the left child (LL)

- Height of x→right→right : h+1
  x→right→left : h or h+1

single rotation with the Right child (RR)

two subtree : h+2, h

- Height of x→left→right : h+1

double rotation with the left child (LR)

▲ BF(x) = +2, BF(x→Left) = −1

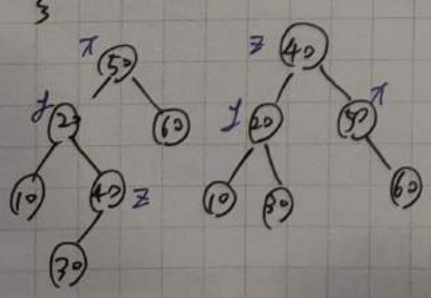
- Height of x→right→left : h+1

double rotation with the right child (RL)

▲ BF(x) = −2, BF(x→Right) = +1

ex: LR Rotation

①
```
nodeType rotateLR1 (nodeType x){
    nodeType y = x->left;
    nodeType z = y->right;
    // RR rotation on y
    y->right = z->left;
    z->left = y;
    x->left = z;
    // LL rotation on x
    x->left = z->right;
    z->right = x;
    return z;
}
```

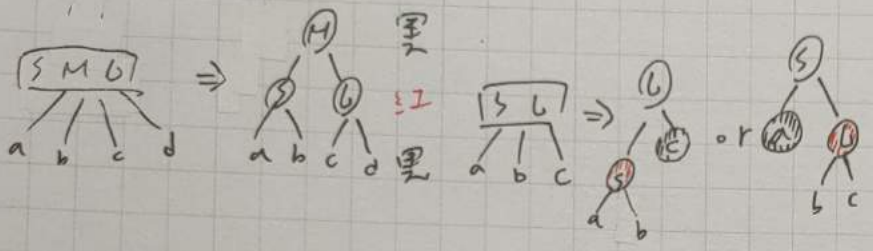


②
```
{
    ⋮
    y->right = z->right;
    x->left = z->right;
    z->right = x;
    z->left = y;
    return z;
}
```

補充：B-tree

♥ Red-black tree = Rotations + 2-3-4 tree + Binary Search Tree

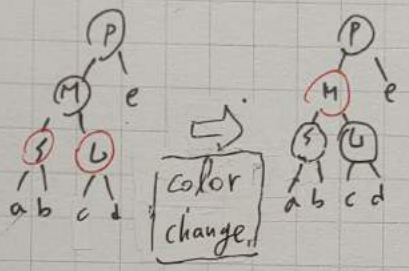
- 3 node, 4 node
- rotations
- 有 2-3-4 tree 的平衡優點，沒有空間浪費的問題
  ▲ easy to keep balanced and simple insertion/deletion
- Every external path has an equal number of Black pointers
- External path cannot have two consecutive Red pointers
- 2-3-4 tree 的高度



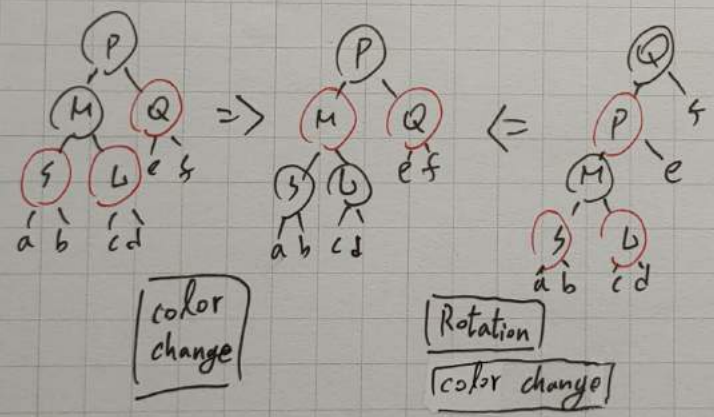
1. Parent is a 2-node

Only change the colors

- to child : red → black
- from parent : black → red



2. Parent is 3-node



♥ Insertion

1. Splits of a node with two red pointers (like the 4-node in a 2-3-4 tree) occur only on the path from the root to a leaf (downward)!
2. Set the pointer to a new-added node as red
3. Rotate if there are two consecutive red pointers

♥ Deletion

1. Find the node to delete, as in a binary search tree
   → two children → swap with the in-order successor
   → Only one child → pointed to by a black pointer
   → Leaf → pointed to by a red or black pointer
2. Replace the node of only one child with its child
3. Delete the leaf if the pointer to it is red
4. Recolor or rotate
   ↳ Leaf pointed to by a black pointer