Ch1 優先佇列

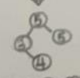
1-01 優先佇列簡介

1、ex: ⑤ ⑥ ③ ④
⇓ pqInsert( )
$\boxed{PQ}$ ⇒ sort Algorithm
⇓ pqDelete( )
③ ④ ⑤ ⑤

1-02 以排序實現優先佇列

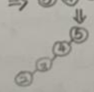
1、selection ⇒ pqInsert( ) O(1)
⇒ pqDelete( ) O(n)
insertion ⇒ pqInsert( ) O(n)
pqDelete( ) O(1)

2、binary search tree ⇒ ⑤ ⑤ ③ ④



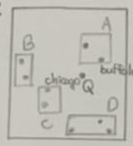
pqInsert O(n)
⇒ pqDelete O(n)

⇒ ⑤ ④ ③ ②



1-03 應用於鄰近搜尋

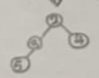
1、ex:



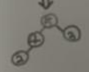
PQ: C, A, D, B
⇓
A, D, Chicago
⇓
Buffalo

1-04 初探資料結構

1、⑤ ④ ③ ②
⇓ pqInsert( ) O(log n)
min-Heap
⇓ pqDelete( ) O(1)



2、⑤ ④ ③ ②
⇓ pqInsert( ) O(log n)
max-Heap
⇓ pqDelete( ) O(1)

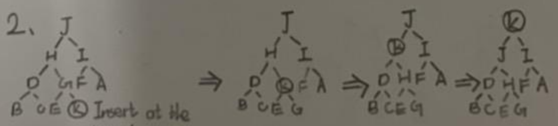


3、① complete binary tree
② value of node is greater (smaller) or equal to the value of the children

1-05 新增資料於堆積結構

1、
```
template<class ItemType>
struct HeapType{
  void ReheapDown(int,int);
  void ReheapUp(int,int);
  ItemType *elements;
  int numElements;
};
```

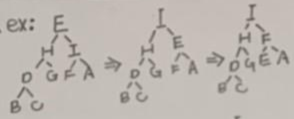
2、



pqinsert( ) O(log n)

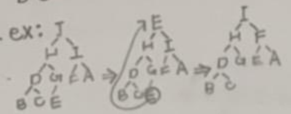
1-06 於堆積結構刪除結點

1. ex:



2. copy the bottom rightmost element to the root
Delete the bottom rightmost node
Fix the heap
P9Delete O(logn)

3. ex:



1-07 應用於霍夫曼編碼

1. Suppose messages are made of letters a, b, c, d and e which appear with probabilities 12%, 4%, 15%, 8%, and 25%, respectively.
2. We wish to encode each character into a sequence of 0s and 1s so that no code for a character is the prefix for another.
3. Using Huffman's algorithm:
a=110, b=1111, c=10, d=1110, e=0

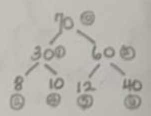
1-08 新增資料於堆疊範例

1.
```
bool heapIsEmpty( );
void heapInsert(HeapItemType& newItem);
void heapDelete(HeapItemType& rootItem);
void heapRebuild(int root);
HeapItemType items[Max_Heap];
int size;
```

2. Insert newItem into the bottom.
NewItem trickles up.

3.

0	70
1	30
2	60
3	8
4	10
5	12
6	40



4.
```
heapInsert(&newItem){
  if(size >= Max_HEAP)
    throw HeapException("Heap Full");
```

1-09 於堆疊刪除資料的範例

1. step1: Return the item in the root
   step2: Copy the item from the last node into the root
   step3: Remove the last node: size
   step4: Transform the semi-heap back into a heap

2. heapRebuild (int root): Psedocode{

```
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]){
    HeapItemType temp = items[root];
    items[root] = items[child];
    items[child] = temp;
    heapRebuild(child);
  } //if()
} //if()
} // heapRebuild()
```

1-10 將陣列直接轉成堆積

1. 由下往上做 ↑

1-11 以指標實作堆積結構

1.

```
typedef struct heapItem{
  int value;
  struct heapItem *parent;
  struct heapItem *leftChild;
  struct heapItem *rightChild;
}heapNode;

typedef enum{LEFT,RIGHT} whichChild;

heapNode *parentOfBottom;
whichChild childAtBottom;
```

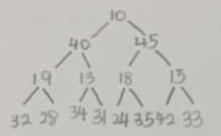
Ch2 堆積變形

02-1 堆積變形初探

1、Double-ended priority Queues (DEPQ)
Min-max Heap , Double-ended Heap (DEAP)
Forest (union) of Heaps
Binomial Heap , Fibonacci Heap

2、Min-max Heap

Level 1 — min
Level 2 — max
Level 3 — min
Level 4 — max

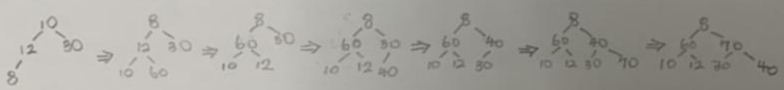


02-2 新增資料於最小最大堆積

1. Decide which level ? min or max
2. Check wether to swap with its parent
   Yes : ReheapUP from its parent
   No : ReheapUP from the current node

02-3 新增資料於最小最大堆積的練習

1、ex: 10 12 30 8 60 40 70



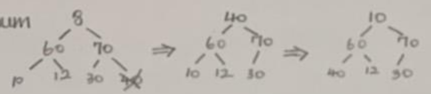
02-4 於最小最大堆積刪除資料

1. Delete smallest
   ⇒ ① Replace the root with the last element
   ② check whether to swap with its smaller child
   No : Reheap Down from the root (recursion)
   Yes : Reheap Down from the root (recursion)

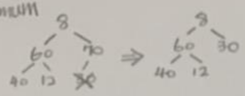
2. Delete Largest
   ⇒ ① Replace the maximum with the last element
   ② Check whether to swap with its larger child
   No : ReheapDown from the current node
   Yes : ReheapDown from the current node

02-5 於最小最大堆積刪除資料的練習

1. minimum



maximum

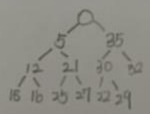


2. level = (int) floor (log2(i+1) % 2) ? Max : Min;
   if((i-1)/2) grandparent = (i-3)/4
   grandchildren = items[i*4+j] for j=3,4,5,6

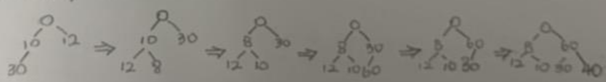
02-6 最小最大堆積的總結

1. Three 4-way trees
   ⇒ max-heap + min-heap + maxheap
   ⇒ each node in maxheap has its parent in min-heap

02-7 新增資料於雙向堆積

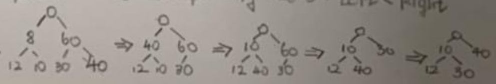


1. Double-ended Priority Queue (DEPQ)
2. Examine the corresponding nodes: Left < Right
   ReheapUp if necessary (recursion)
3. ex: 10, 12, 30, 8, 60, 40



02-8 於雙向堆積刪除資料

1. Smallest
   ⇒ Replace the root of min-heap with the last element
   ReheapDown if necessary
   Examine the corresponding nodes: Left < Right
2. Largest
   ⇒ Replace the root of max-heap with the last element
   ReheapDown if necessary
   Examine the corresponding nodes: Left < Right
3、



02-9 雙向堆積的總結

1. Two heaps
   ⇒ Pseudo root + min-heap + max-heap
   ⇒ Each node in max-heap corresponds to one in min-heap

02-11 堆積結構的應用

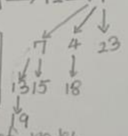
1. external sort ⇒ Large amount of data on secondary storage
   e.g. quick + heap sort

2. Merge of priority queues
   ⇒ Two cooks → one cook
   multiple severs: job queues

02-12 可合併的堆積結構

1. A binomial heap is a collection of binomial trees that satisfy the heap property and have distinct orders
   ⇒ Two binomial trees of the same order can be merged

2. example:
   $9=2^0+2^3$ head[H_1] → 12 → 1



any number $=a_0 2^0+\dots+a_k 2^k$ where $a_0,\dots a_k$ are 0 or 1

Given the number of nodes
⇒ a unique structure

3. 13 node look like?
   $13=(1101)_2=2^3+2^2+0+2^0$
   26 node look like?
   $26=(11010)_2=2^4+2^3+0+2^1+0$

02-13 二項式堆積

1. A linked list sorted by the orders of binomial trees (degrees of the roots)
   Merge two binomial trees of the same order (from left to right)

2. Insert: ① Insert into the linked list of the roots
   ② call merge function

3. Delete: ① Find the minimum from the linked list of the roots
   ② Delete the root having the minimum
   ③ Add its children into the linked list
   ④ Call merge function

02-14 堆積的總結

1. Fibonacci Heap: Definition
   ① Doubly linked list on the siblings (tree roots)
   ② Doubly linked list between parent and child
   ③ Merge: simply concatenate two lists of tree roots

Ch3 由下而上的成長的平衡二元樹

03-1 搜尋的基本觀念

1、Use the search key

03-2 搜尋機制的基本運算

1、insert, delete, retrieve

2、① Linear implementation : 4 categories // appropiate small table or unsort with few deletions
   Unsorted ⇒ array based, pointer based
   sorted ⇒ array based, pointer based
   ② Nonlinear implementation : // balanced binary tree
   Binary search tree ⇒ several advantage over linear

03-3 實作方式的不同選擇

1、Unsorted-array
   ① insert O(1)
   ② deletion O(n)
   ③ retrieval O(n)

2、sorted-array
   ① insert / deletion O(n)
   ② binary search O(logn)

Traversal of four is O(n)

3、Usorted-pointer
   ① no data shifts
   ② insertion O(1)
   ③ deletion / retrieval O(n)

4、sorted-pointer
   ① no data shifts
   ② insertion / deletion / retrieval O(n)

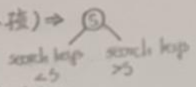
03-4 初探平衡二元樹

1、① 2-3 tree, 2-3-4 tree
   ② AVL tree, Red-black tree

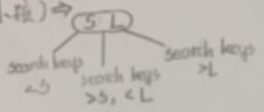
2、2-3 tree ⇒ ① general tree, not binary trees
   ② never taller than minimum-height binary tree
   never has height greater than $[\log_2(n+1)]$

03-5 簡介2-3樹

1. ① 2-node (2個小孩) ⇒

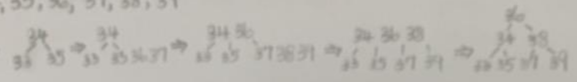


② 3-node (3個小孩) ⇒



2. insert 33, 34, 35, 36, 37, 38, 39

⇒ 33, 34, 35 ⇒



03-6 2-3樹的樹高

1. 2-3樹 worst-case 為二元搜尋樹最好情況

03-7 新增資料於2-3樹的演算法

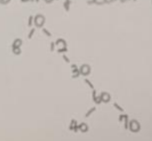
1.
```
insertItem(in ttTree, in newItem)
  Locate and add newItem to leafNode;
  if (leafNode now has three items)
    split(leafNode);
```

2.
```
split(inout treeNode)
  if (treeNode == root)
    creat a new node p;
  else
    p = parent of treeNode;
  Replace treeNode with node1 and node2, so that p is their parent;
  Place the smallest and largest key into node1 and node2;
  if (treeNode is not a leaf)
    node1 = parent of two leftmost children under treeNode;
    node2 = parent of two rightmost children under treeNode;
  Move the middle key up to p;
  if (p now has three items)
    split(p);
```

03-8 新增資料於2-3樹的範例

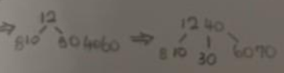
1. insert 10, 12, 30, 8, 60, 40, 70

① binary search tree



② 2-3 tree

10 12 30 ⇒



03-10 於23樹刪除資料的演算法

1、Locate the leaf at which the search for I would terminate

2. Delete I from the leaf

3. If the leaf now contains one item, you are down

4. If the leaf now contains no item, choose one of the following operations to fix

   (a) Redistribute the value: retain the tree structure

   (b) Merge into a leaf: its parent has one less child

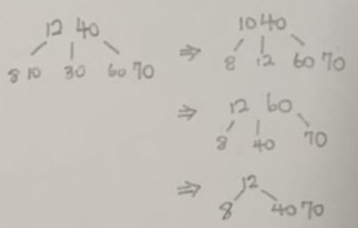
5、deleteItem (in ttTree, in theKey)

```
X = the tree node whose search key equals thekey;
if (X is not a leaf)
Y = Successor(X);
swapKey (X, Y);
X = Y;
Delete thekey for X
if (X now has no item)
  fix(X);
```

```
fix(in X)
  if(X==root)
    remove the root;
  else
    P = parent of X;
    if( the nearest sibling of X has two items)
      Redistribute items among the sibling, P and X;
      if(X is not a leaf)
        Move appropriate child from sibling to X;
    else
      s = the nearest sibling of X;
      Move appropriate itemdown from p to s;
      if ( X is not a leaf)
        Move X's child to s;
      remove X;
      if(p now has no item)
        fix(p);
```

03-11 於23樹刪除資料的範例

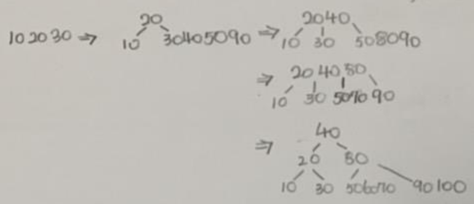
1ex: 30 10 60



03-12 新增資料於234樹

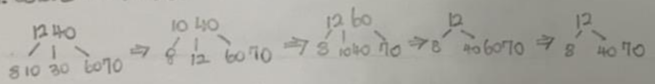
1. are general trees, not binary trees
   never taller than 2-3樹
   extensions of 2-3樹

2. ex: insert 10, 20, 30, 40, 50, 90, 80, 70, 60, 100



03-13 於234樹刪除資料

1. Locate the node n that contains the item theItem
2. Find theItem's inorder successor and swap it with theItem (deletion will always occur at a leaf)
3. If that is a 3-node or 4-node, remove theItem
4. Delete: 30, 10, 60



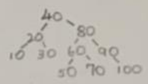
03-14 23樹和234樹的總結

1. 234 tree to 23 tree more than one data, more than one children
2. 234 tree are more efficient than the corresponding algorithms for 2-3 tree

Ch4 由上而下成長的平衡二元樹

04-1 複習二元搜尋樹

1. insert 40, 20, 10, 80, 90, 100, 30, 60, 50, 70



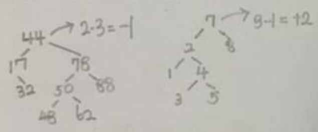
2. delete ⇒ Located node M that is easier to deleted
   →M is the leftmost node in X's right subtree
   →M will have no more than one child (0 or 1): easier
   →M's key is called the inorder successor of X's key
   Copy the item that is in M to X
   Remove the node M from the tree

04-1 AVL樹的原理

1. ① A balanced binary search tree
   ② minimum-height binary search tree
2. check whether the tree is balanced, if unbalanced rotate it
3. Balance Factor (BF)
   BF(a node) = |h(left subtree) − h(right subtree)|
   BF is no more than 1

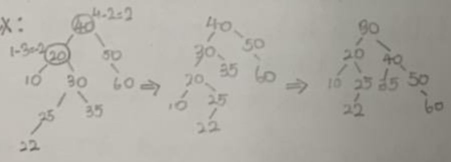
04-2 AVL樹的平衡係數

1.



04-3 AVL樹的旋轉

1. single rotation, double rotation
2. ex:



04-4 AVL樹的單一旋轉

1. Let x be the node at which x->left and x->right differ more than 1

   (LL)
   BF(x)=2 BF(x->left)=+1 or 0

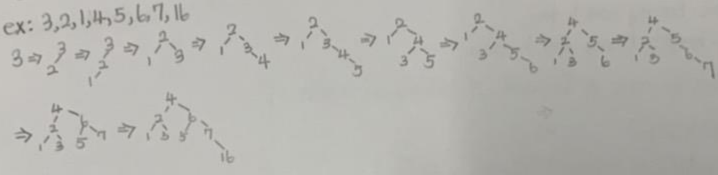
   (RR)
   BF(x)=-2 BF(x->right)=-1 or 0

2. ex:

```
nodeType rotateLL(nodeType x){
    nodeType y=x->left;
    x->left=y->right;
    y->right=x;
    return y;
}
```

```
nodeType rotateRR(nodeType x){
    nodeType y=x->right;
    x->right=y->left;
    y->left=x;
    return y;
}
```

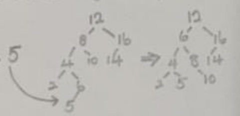
3. ex: 3,2,1,4,5,6,7,16



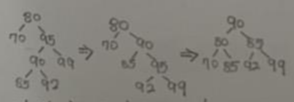
04-5 AVL樹的複式旋轉

1. (LR): RR⇒LL
   BF(x)=+2, BF(x->left)=-1
   (RL): LL⇒RR
   BF(x)=-2, BF(x->right)=+1

2. ex:
   (LR) insert 5



ex:
(RL) insert
85 or 92



04-6 實作AVL的複式旋轉

1.
```
nodeType rotateLR(nodeType x){
    x->left=rotateRR(x->left);
    return rotateLL(x)
}
```

2.
```
nodeType rotateRL(nodeType x){
    x->right=rotateLL(x->right);
    return rotateRR(x);
}
```

04-7 刪除AVL樹的資料

1. Delete a node x as in a binary search tree and the last node deleted is a leaf
2. Trace the path from the new leaf towards the root
3. After we perform a rotation at X, we may have to perform a rotation at some ancestor of X

04-8 比AVL樹和234樹

1. An AVL tree is a binary search tree that is guaranted to remain balansed using rotations.
2. A 2-3-4 tree the variant of a binary search tree in which nodes can contain more one data and have more than two children.

04-9 紅黑數的原理

1、A binary search tree to represent a 2-3-4 tree
   Rotation like AVL tree
   Easy to keek balanced and simple insertion / deletion

2、class RBTreeNode {
```
    TreeItemType    item;
    RBTreeNode *    leftChildPtr, rightChildPtr;
    colorType       leftColor, rightColor;
```

04-10 比較紅黑樹和234樹

1、Every external Path has an equal number of black pointers → Height of 234 tree

2、External path cannot have two consecutive red pointers.

04-11 紅黑樹的分裂

1 parent 2-node ⇒ change colors
   ⇒ pointers to and form its parents are black

parent 3-node ⇒ color change (LL Rotation) (RR Rotation)
   ⇒ pointers sibling、parent is red

04-12 新增資料於紅黑樹

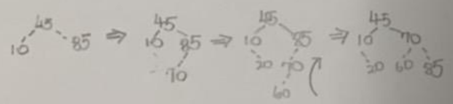
1、splits of a node with two red pointers, occur only on the path from the root to a leaf

2、set the pointer to a new-added node as red

3、Rotate if there are two consecutive red pointers

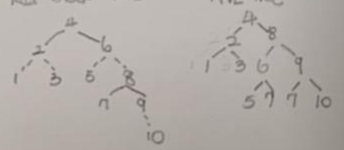
04-13 新增資料於紅黑樹的範例

1、ex: 45, 85, 10, 70, 20, 60



04-14 比較紅黑樹和AVL樹

1、Input 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
   Red-black Tree   AVL Tree



04-15 刪除紅黑樹的資料

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

04-16 於紅黑樹刪除資料的各種狀況

1. pointer pointing to its parent red?
   Yes, recolor or rotation
2. pointer pointing to its sibling red?
   Yes, rotations + recolor
3. sibling has any child?
   Yes, rotation + recolor
   No, upward recursion

04-17 平衡數的總結