Runtime

$O(n) \geq$

$\Theta(n) =$

$\Omega(n) \leq$

Amortized Time

c_i, a_i are costs

Operations $c_0, c_1, c_2 \cdots \quad c_i \in O(f(i))$

If exists $a_0, a_1, a_2 \cdots \quad a_i \in O(g(i))$

and $\sum_{0 \leq i \leq k} a_i \geq \sum_{0 \leq i \leq k} c_i \quad (\forall k)$

then the c operations run in $O(g(i))$ amortized Time

Hash

① Same Object, same hash

② Try to avoid collision

$h(\text{string}) = s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} \cdots + s_{n-1} \pmod{2^{32}}$ ← Int.MAX.

Hashmap

Load Factor $L = N/M$ (N: # of items, M: # of buckets)

Probing

① Linear try $h(K)+m,\ h(K)+2m \cdots$

② Quadratic $h(K)+1 \cdot m,\ h(K)+2^2 \cdot m,\ h(K)+3^2 \cdot m \cdots$

③ Double hashing $h(K)+h'(K),\ h(K)+2h'(K) \cdots$

Views

[start, end)

List<T>.Sublist(start, end)

Map<k, v>.keySet(); values(), .entrySet()

keySet() → Set<k>

values() → Collection<v>

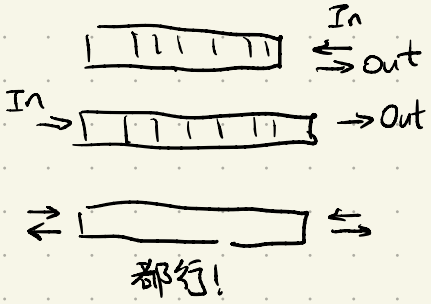
entrySet() → Collection<Map.Entry<k, v>>

List Types

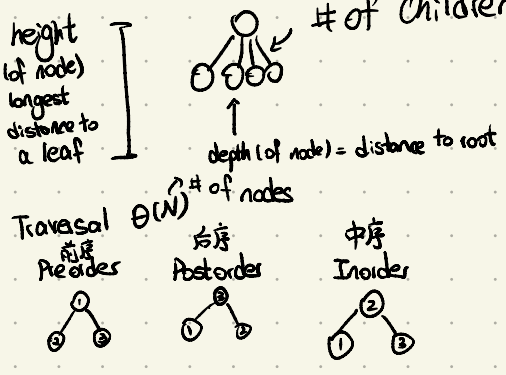
Stack LIFO

Queue FIFO

Deque

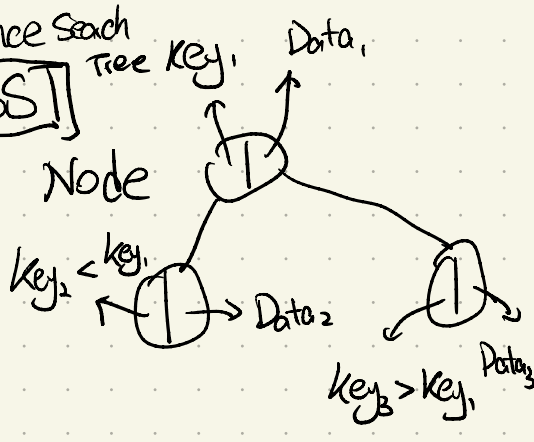


TREES



Balance Seach Tree

BST



Quad Tree 四叉树

2D space into quadrants.

PR Quad Tree

≤2 points per leaf.

Heap —Implements→ Priority Queues

↓

max heap → "add", "find max", "remove max"

最大在root.

heaps can be "nearly complete"

Insert: Insert at bottom, re-heapify-up (往上換)

Remove: Pick Left-subtree rightmost node, exchange with root. delete root & bubble down.

```java
/** Apply WHATTODO to all labels in T that are >= L and < U,
 *  in ascending natural order. */
static void visitRange(BST<String> T, String L, String U,
                       Consumer<BST<String>> whatToDo) {
  if (T != null) {
    int compLeft = L.compareTo(T.label ()),
        compRight = U.compareTo(T.label ());
    if (compLeft < 0)                   /* L < label */
        visitRange (T.left(), L, U, whatToDo);
    if (compLeft <= 0 && compRight > 0) /* L <= label < U */
        whatToDo.accept(T);
    if (compRight > 0)                  /* label < U */
        visitRange (T.right (), L, U, whatToDo);
  }
}
```

height # of data in range

$\in O(h+M)$

Generic

static <T> List<T> emptyList()

class ABC<T>

<T extends List<Collections<String>>>

<? extends xx>

↓

wildcard

Int.

Type	Bits	signed
byte	8	Y
short	16	Y
char	16	N
int	32	Y
long	64	Y

Bit op.

and (mask) a & b

or (set) a | b

flip (xor) a ^ b

flip all (not) ~a

Shift

logical left a << b

right a >>> b

sign-extended a >> b

Sorting

Internal ⇒ {_ _ _ _ _} keeps data in main mem

External ⇒ {_ _} + {_ _} large amounts of data in batches

Insertion

empty sequence, insert into output

$\underbrace{\Theta(k)}_{\text{for one item}}$ ⇒ $O(N^2)$

Inversions

good for nearly sorted

Inversions

- Can run in $\Theta(N)$ comparisons if already sorted.
- Consider a typical implementation for arrays:

```
for (int i = 1; i < A.length; i += 1) {
  int j;
  Object x = A[i];
  for (j = i-1; j >= 0; j -= 1) {
    if (A[j].compareTo(x) <= 0)  /* (1) */
      break;
    A[j+1] = A[j];               /* (2) */
  }
  A[j+1] = x;
}
```

6 3 5 7 8 2 4

A[j] 往后推

A[j] 插到最前 (比3小的数保留原位)

- #times (1) executes for each j ≈ how far x must move.
- If all items within K of proper places, then takes $O(KN)$ operations.
- Thus good for any amount of *nearly sorted* data.
- One measure of unsortedness: # of *inversions*: pairs that are out of order (= 0 when sorted, $N(N-1)/2$ when reversed).
- Each execution of (2) decreases inversions by 1.

ShellSort

Example of Shell's Sort

	Inversions Left #I	Cumulative Comparisons #C
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	120	0
0 14 13 12 11 10 9 8 7 6 5 4 3 2 1 15	91	1
0 7 6 5 4 3 2 1 14 13 12 11 10 9 8 15	42	11
0 1 3 2 4 6 5 7 8 10 9 11 13 12 14 15	4	31
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0	50

I: Inversions left.
C: Cumulative comparisons used to sort subsequences by insertion sort.

Heap Sort

heapify

从 Index/2 (倒数第二层开始往下)

re-heapify (bubble down)

Merge Sort

break into 2 parts, recursively sort & merge

Merge Sorting 合并排序

Idea: Divide data in 2 equal parts; recursively sort halves; merge results.

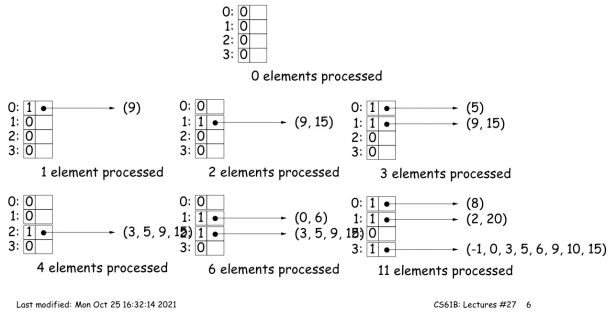
- Already seen analysis: $\Theta(N \lg N)$.
- Good for *external sorting*:
  - First break data into small enough chunks to fit in memory and sort.
  - Then repeatedly merge into bigger and bigger sequences.
- Can merge K sequences of *arbitrary size* on secondary storage using $\Theta(K)$ storage:

```
Data[] V = new Data[K];
For all i, set V[i] to the first data item of sequence i;
while there is data left to sort:
    Find k so that V[k] is smallest;
    Output V[k], and read new value into V[k] (if present).
```

Illustration of Internal Merge Sort

For internal sorting, can use a *binomial comb* to orchestrate:

L: (9, 15, 5, 3, 0, 6, 10, -1, 2, 20, 8)



QuickSort

pivot = Median(0, $\frac{n}{2}$, n)

size ≤ 2 : insertion sort.

worst $\Theta(N^2)$, best $\Omega(N \lg N)$

avg $\Theta(N \lg N)$

Radix Sort $\Theta(B)$

LSD ← ↓ total # of key

MSD →

↳ keep lists of each step seperate

Distribution Counting

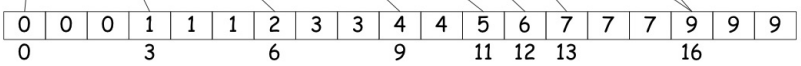
Distribution Counting Example

- Suppose all items are between 0 and 9 as in this example:

7	0	4	0	9	1	9	1	9	5	3	7	3	1	6	7	4	2	0

3	3	1	2	2	1	1	3	0	3	*Counts*
0	1	2	3	4	5	6	7	8	9	

0	3	6	7	9	11	12	13	16	16	*Running sum*
< 0	< 1	< 2	< 3	< 4	< 5	< 6	< 7	< 8	< 9	



0	0	0	1	1	1	2	3	3	4	4	5	6	7	7	7	9	9	9
0			3			6			9		11	12	13			16		

- "Counts" line gives # occurrences of each key.
- "Running sum" gives cumulative count of keys $<$ each value...
- ...which tells us where to put each key:
- The first instance of key k goes into slot m, where m is the number of key instances that are $< k$.