Non-Comparison Sort

Counting Sort

- Count the number of each number.

1. Allocate an array `count[k+1]`.
2. Scan array A. For i=1 to `N`, increment `count[A[i]]`.
3. Scan array `count`. For i=0 to `k`, print `i` for `count[i]` times.

1. Allocate an array `C[k+1]`
2. Scan array A. For i=1 to `N`, increment `C[A[i]]`
3. For `i=1` to `k`, `C[i]=C[i-1]+C[i]`
   - *`C[i]` now contains number of items less than or equal to `i`*
4. For `i=N` downto `1`, put `A[i]` in new position `C[A[i]]` and decrement `C[A[i]]`

- Time complexity: $\mathcal{O}(n+k)$

Bucket Sort

- Distribute the records by keys into appropriate buckets
- Sort each bucket by comparison sort.

Time complexity:

Assume items are uniformly distributed. Totally n elements, range $[0, k]$, into $\frac{n}{c}$ buckets. Then each bucket contains c elements. Then totally time complexity:
$\mathcal{O}(\frac{n}{c} \times c \log c) = \mathcal{O}(n \log c) = \mathcal{O}(n)$.

Radix Sort

Procedure:

From the least significant bit to the most siginificant bit, do bucket sort. In i_{th} round, sort in each bucket by the least i bits.

Actually, in each round, you just need to scan through the result array in the last round, and directly insert into the corresponding bucket, then the elements in each bucket is already sorted according to the least i bits.

Correctness: Induction.

Time complexity: $\mathcal{O}(n \log k)$

Linear Time Selection

The selection problem

Find i_{th} smallest element in the array.

Randomized selection algorithm

Basic idea comes from quick sort.

We only care about the part containing the target element.

Recursion !

```
Rselect(int A[], int n, int i) {
// find i-th smallest item of array A of size n
  if(n == 1) return A[1];
  Choose pivot p from A uniformly at random;
  Partition A using pivot p;
  Let j be the index of p;
  if(j == i) return p;
  if(j > i) return Rselect(1st part of A, j-1, i);
  else return Rselect(2nd part of A, n-j, i-j);
}
```

Time complexity: $\mathcal{O}(n)$

Deterministic selection algorithm

In worst case, time complexity degenerates to $\mathcal{O}(n^2)$.

A pivot close to median is better.

Use median of median to find a **approximate median** as pivot.

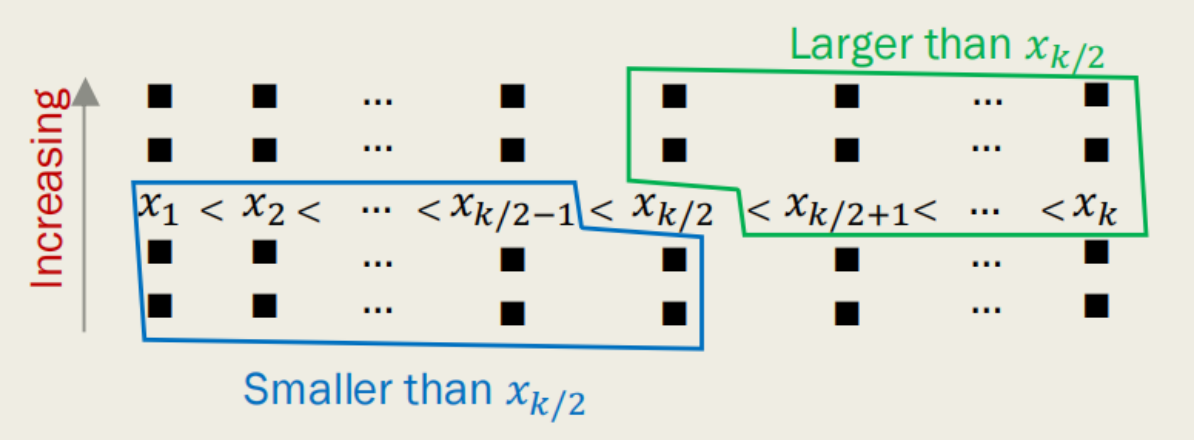
p here is not exactly the median of original sequence.

```
Dselect(int A[], int n, int i) {
// find i-th smallest item of array A of size n
  if(n == 1) return A[1];
  Break A into groups of 5, sort each group;
  C = n/5 medians;
  p = Dselect(C, n/5, n/10);
  Partition A using pivot p;
  Let j be the index of p;
  if(j == i) return p;
  if(j > i) return Dselect(1st part of A, j-1, i);
  else return Dselect(2nd part of A, n-j, i-j);
}
```

Choose Pivot

Same as Rselect

To calculate the time complexity, we need to find the possible range of our pivot.



$T(n) = O(n) + T(\frac{n}{5}) + T(\frac{7n}{10})$

Hash

A dictionary that can insert and find key in $\mathcal{O}(1)$.

$key \overset{hash\ function}{\longrightarrow} A \overset{compression\ map}{\longrightarrow} index$

conllision

Separate Chaining

Open addressing

- Linear Probing : $h_i(x) = (h(x) + i)\%n$
- Quadratic Probing : $h_i(x) = (h(x) + i^2)\%n$
- Double Hashing : $h_i(x) = (h(x) + i * g(x))\%n$

If table size is not a prime number, sometimes we might never find an empty slot.

Load factor

$\frac{\#objects\ in\ hash\ table}{\#buckets\ in\ hash\ table}$

remove

separate chaining: easy

open addressing: lazy deletion

Rehash

Amortized time

Bloom filter

False negative and false positive

Binary search tree

Definition: for each node, its key is greater than all nodes in left subtree and less than all nodes in right subtree.

Operations:

search / insert / remove / predecessor / successor / rank search

all $\mathcal{O}(\log n)$

range search : $\mathcal{O}(n)$

search:

```
node *search(node *ptr, Key k) {
  if(ptr == NULL) return NULL;
  if(k == ptr->item.key) return ptr;
  if(k < ptr->item.key)
    return search(ptr->left, k);
  else return search(ptr->right, k);
}
```

insert:

```
void insert(node *&root, Item item)
// EFFECTS: insert the item as a leaf,
// maintaining the BST property.
{
  if(root == NULL) {
    root = new node(item);
    return;
  }
  if(item.key < root->item.key)
    insert(root->left, item);
  else if(item.key > root->item.key)
    insert(root->right, item);
}
```

remove x:

- Leaf : directly remove.
- Degree-one : directly connect x's son with x's father.

- Degree-two : find the predecessor / successor y in the subtree. Replace x by y, remove y, since y only have one son, it's simple.

predecessor of x

- left subtree of x is not empty : same as remove
- empty : keep find father until get a key less than x. (Which means find a left father).

successor of x

- right subtree of x is not empty : same as remove
- empty : keep find father until get a key larger than x. (Which means find a right father).

rank search

Consider find k_{th} elements in a subtree.

Record the size of each subtree.

Compare the k with the left size

- <= leftsize , recursion in left subtree
- = leftsize + 1, return root
- otherwise, recursion in right subtree

range search

As long as the search range cover all or part of one subtree, recursion (brute-force to find all possible elements).

Add root into result if it's within the range.