Lectures 24-25

Odd Sorts and Direct Address Tables

Lectures 24-25

Today:

- Alternate Comparison Sorting Algorithms
- Lower Bound on Comparison Sorting
- Sorting in Linear Time
- Direct Address Tables

Alternate Sorts: Tree Sort

The Usual Problem: Sort an array of *N* items.

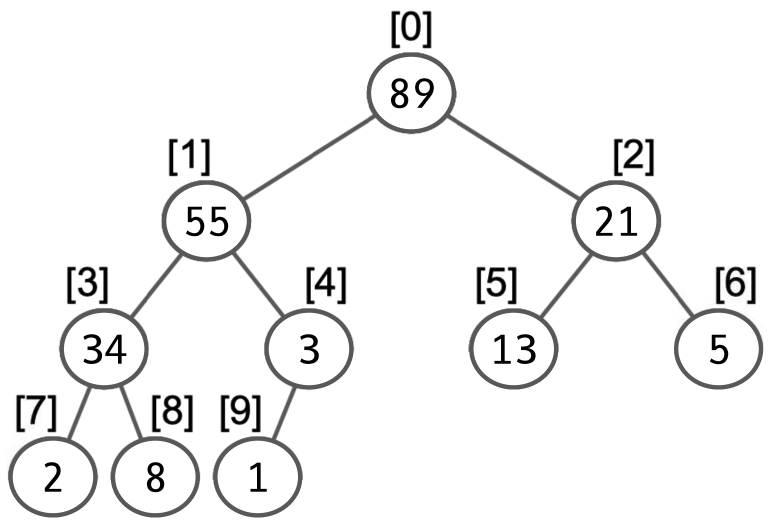
Strategy: Insert all *N* into a binary search tree.

53	77	22	59	79	43	54	18	29	92

Alternate Sorts: Heap Sort

Strategy:

- build a
- call
    -



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
89	55	21	34	3	13	5	2	8	1

Lower Bound on Comparison Sorting

Q. Can you ever do better than $O(N \log N)$?

Information Theory: In the best case, a single comparison can

Q. How big is the solution space?

Therefore, any solution will cost at least

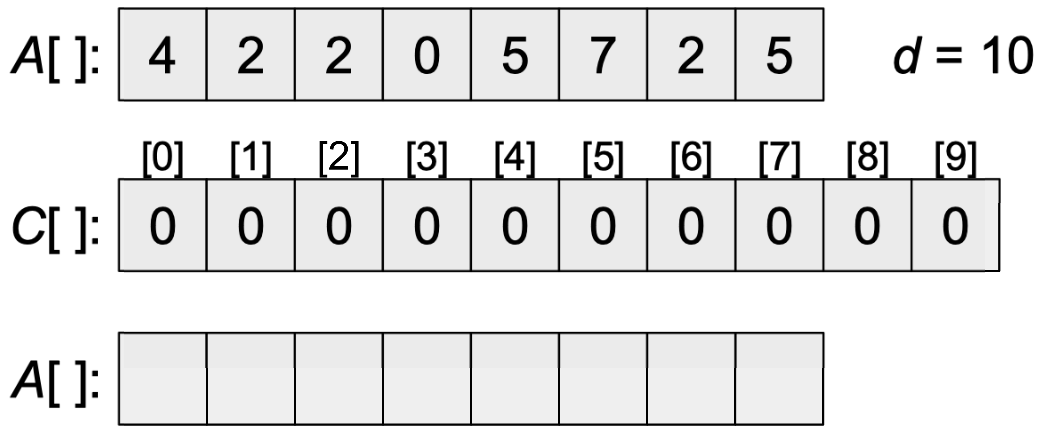
Sorting Integers in Linear Time?

Assume integers in range

Strategy:

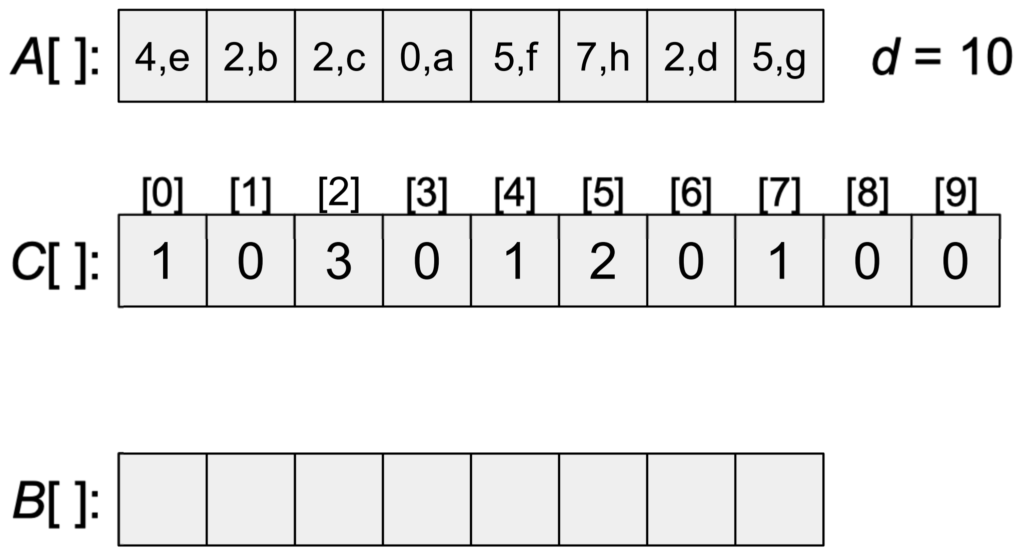
- take advantage of the fact that arrays have $O(1)$ *direct access*.
- count frequency of each element in $A[N]$
- place elements by frequency / rank

Algorithm:



Counting Sort — Stable Version

```
// Desc: src array A[n], dest array B[n], place sorted A[] into B[]
//  Pre: Each obj contains .key in range [0,d-1]
void countingSort(obj A[], obj B[], unsigned n, unsigned d) {
    int C[d];


    for (int i = 0; i < d; i++)
        C[i] = 0;

    for (int j = 0; j < n; j++)
        C[A[j].key]++;







}
```



Analysis of Counting Sort

Running time of either version is

- if *N* ~ *d*, then

But how big can *d* be in practice?

-
- if sorting 32-bit integers, then
- another bad *d*:

Some objects don't have a range:

- E.g., What's *d* for
-

Radix Sort

Problem: Sort a set of [birth-]dates.

Strategy: Use Counting Sort on each “digit” starting with the least significant digit

- ties are broken by stability

2004/05/29

2001/11/29

2004/05/16

2003/05/29

2003/05/14

2002/02/14

2002/09/01

Dynamic Set ADT — Revisited

Same idea extends to Dynamic Set

Assumption: all keys are unique

Strategy:

- use array A[0…d-1] to store pointers
- NULL pointer means empty

If there is no associated data -> bit vector

- a large Boolean array

```
obj * A[d];

.insert(obj * x) {
    A[x->key] = x;
}

.search(T key) {
    return A[key];
}

.delete(T key) {
    A[key] = NULL;
}
```

Sample Optimization (E.g., from CMPT 295)

```c
int str_alnum(char *s) {
   int i;
   for (i = 0; i < strlen(s); i++) {
      if (!isalnum(s[i])) {
         return 0;
      }
   }
   return 1;
}
```

`bool isalnum(char c);`

Strategy: Range of `char` is

- Pay for an implementation that is a single instruction!

```
char vec[256] = {
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...};

bool isalnum(char c) {
     return vec[c];
} // isalnum
```

Direct Address Tables — Large *d*

Size of table depends on

- E.g., SFU ID# 10^9 table entries

but # of students ~ 10^6 => only 0.1% full

What if there is no range for keys?

- E.g.,

Strategy: craft a function h(x)

- map key space -> address space
- [0…d-1] -> A[0…m-1]
- Choose *m* so that a ~ 1
- Design *h*(*x*) to randomly distribute keys
- But what if h(x) = h(y)? — a collision?

```
obj * A[m];

.insert(obj * x) {
        A[h(x->key)] = x;
}
.search(T key) {
        return A[h(key)];
}
.delete(T key) {
        A[h(key)];
}
```