Sorting

Fall 2020

School of Software Engineering
South China University of Technology

“Linear-Time” Sorts

Bucket Sort

```
for (i=0; i<n; i++)
   B[A[i]] = A[i];
```

- Bucket Sort
  - The key values are used to determine the positions for the records in the final sorted array.
  - It works only for a permutation of the numbers from 0 to N-1.
  - The cost is $\Theta(N)$ time regardless of the initial ordering of the keys.

Bucket Sort

```
/**
* Bucket Sort
* Allow for duplicate values among keys
* Allow for a set of N records falling in a range larger
* than N ([0,MaxKeyValue-1])
**/
template <typename E, class getKey>
void binsort(E A[], int n) {

 List<E> B[MaxKeyValue]; //An array of linked lists
 E item;

 //assign records to bins
 for (i=0; i<n; i++)
  //All records with key value i are placed in bin B[i]
  B[getKey::key(A[i])].append(getKey::key(A[i]));

 //process MaxKeyValue bins to output records
 for (i=0; i<MaxKeyValue; i++)
  for (B[i].setStart(); B[i].getValue(item); B[i].next())
   output(item);
}
```

Bucket Sort

- The time cost consists of
  - $\Theta(N)$ for assigning N records to bins.
  - Scan **`MaxKeyValue`** bins to output N records
    - If **`MaxKeyValue` is** $\Theta(N)$, the total cost is $\Theta(N)$;
    - If **`MaxKeyValue`** is $\Theta(N^2)$, the total cost becomes $\Theta(N+N^2)=\Theta(N^2)$;
- A large key range requires an unacceptably large array B.
  - Useful only for a limited key range.

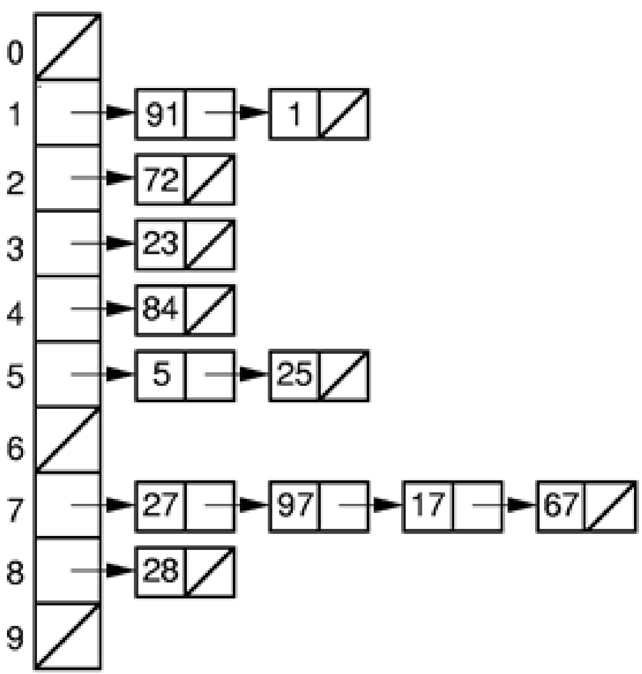
Bucket Sort

- A further generalization
  - Each bin is associated with a range of key values, instead of a single key value
- A bucket sort assigns records to buckets and then relies on some other sorting technique to sort the records within each bucket.
  - A small number of records will be put in each bucket by relatively inexpensive bucketing process
  - A cleanup sort within the bucket will be relatively cheap.

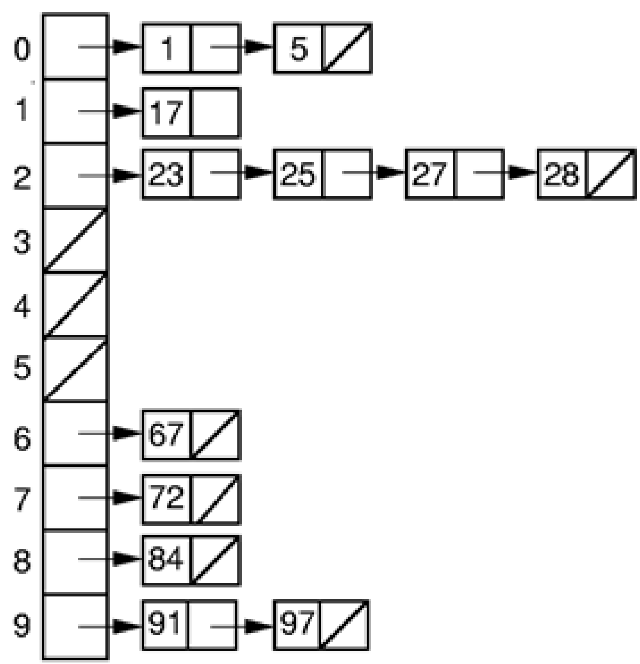
Bucket Sort

- Consider a sequence of records with keys in the range 0 to 99
  - **27,91,1,97,17,23,84,28,72,5,67,25**
- There are 10 buckets available
- We can assign records to buckets by taking key%10.

B[A[i]%10]



B[A[i]/10]



91,1,72,23,84,5,25,27,97,17,67,28

1,5,17,23,25,27,28,67,72,84,91,97

Bucket Sort

- In the previous example,
  - There are b=10 buckets and N=12 keys;
  - The key values are in the range of 0 to b^2-1;

- The records are assigned to buckets based on the keys' digit values working from the rightmost digit to the leftmost.
  - Round 1: the bin number is **`B[A[i]%10]`**
  - Round 2: the bin number is **`B[A[i]/10]`**
  - Finally, the records are taken from the buckets *in order* which produces a sorted list

- The time cost is $\Theta(N)$.

Radix Sort

- Radix Sort – assign records to buckets with the buckets computed based on the radix or the base of the key values
  - It would work for any number of buckets
    - base = 10, 2, 8, 16, ...
  - It can be extended to any number of keys in any key range
  - Assign records to buckets based on the keys' digit values working from the rightmost digit to the leftmost.

Radix Sort

- Example: the number of buckets is 8, and a key is 999
  - Round 1: 999%8 = 7,   key 999 -> bucket 7
  - Round 2: (999/8)%8 = 4,   key 999 -> bucket 4
  - Round 3: $(999/8^2)\%8 = 7$,   key 999 -> bucket 7
  - Round 4: $(999/8^3)\%8 = 1$,   key 999 -> bucket 1
  - Convert 999 to an octal number: $999 = 1*8^3+7*8^2+4*8+7$ = 0x1747

  - Convert MaxKeyValue to a number with b(number of buckets) as the base
  - If the converted MaxKeyValue has r digits, all the keys are assigned to buckets r times
  - Keys are taken from the buckets in order and reassign to the buckets for the next round.

- The running time is O(r(N + b))
  - r is the number of passes,
  - N is the number of elements to sort,
  - b is the number of buckets.

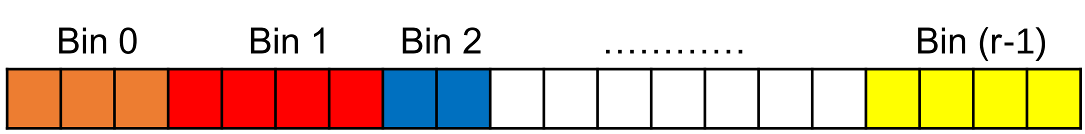
Radix Sort

```
/*
* Radix sort an array of Strings.
* Assume all characters are ASCII, residing in the first 256
* positions of the Unicode character set.
* Assume all have same length(stringLen).
*/
void radixSortA( vector<string> & arr, int stringLen ){
   const int BUCKETS = 256;
   vector<vector<string>> buckets( BUCKETS );

   for( int pos = stringLen - 1; pos >= 0; --pos ){
      for( string & s : arr )
         //Adds s at the end of the buckets[ s[ pos ] ]
         buckets[ s[ pos ] ].push_back( std::move( s ) );

      int idx = 0;
      for( auto & thisBucket : buckets ){
         for( string & s : thisBucket )
            arr[ idx++ ] = std::move( s );

         thisBucket.clear( );
      }
   }
}
```

Radix Sort

- How to implement the Radix Sort efficiently?
  - The number of keys assigned to a bucket can be greater than one and may be as large as the total number of keys
    - Let a bucket points to an array with size equal to the number of keys?
    - Let a bucket points to a linked list of keys?
    - Neither is good!

- The total number of keys in all the buckets is known!
  - Reserve an array with the size of the total number of keys
  - Get to know the number of keys in each bucket



Radix Sort (V)

```
/*
* Counting radix sort
* B[] is array for buckets
* cnt[i] stores numbers of records in bucket[i]
* b is numbers of buckets(base), r is number of passes
*/
template <typename E, typename getKey>
void radix(E A[], E B[], int n, int r, int b, int cnt[]) {

  int j;

  for (int i=0, btoi=1; i<r; i++, btoi*=b) {//for r digits
    for (j=0; j<b; j++) cnt[j] = 0;

    //Count # of records for each buckets on this pass
    for(j=0; j<n; j++) cnt[(getKey::key(A[j])/btoi)%b]++;

    //Index B: cnt[j] will be index for last slot of bucket j.
    for (j=1; j<b; j++) cnt[j] += cnt[j-1] ;

    /*Put records into buckets, from bottom of each
  bucket.*/
    for (j=n-1; j>=0; j--)
      B[--cnt[(getKey::key(A[j])/btoi)%b]] = A[j];

    for (j=0; j<n; j++) A[j] = B[j]; //Copy B back to A.
  }
}
```

Radix Sort

Array A[]

27	91	1	97	17	23	84	28	72	5	67	25

Count in 1st pass

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

Index positions for B[]

0	1	2	3	4	5	6	7	8	9
0	2	3	4	5	7	7	11	12	12

Array A[] in the end of 1st pass

91	1	72	23	84	5	25	27	97	17	67	28

```
B[--cnt[(getKey::key(A[j])/btoi)%b]] = A[j];
```

Radix Sort

Array A[] in the end of 1st pass

91	1	72	23	84	5	25	27	97	17	67	28

Count in 2nd pass

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

Index positions for B[]

0	1	2	3	4	5	6	7	8	9
2	3	7	7	7	7	8	9	10	12

Array A[] in the end of 2st pass

1	5	17	23	25	27	28	67	72	84	91	97

B[--cnt[(getKey::key(A[j])/btoi)%b]] = A[j];

Radix Sort

- Time complexity analysis
  - It requires r passes over the list of n numbers in base b, with Θ (N + b) work done at each pass
  - The total cost is Θ (r(N + b))

- How do r, b, and N relate?
  - The base b is usually a small number.
    - e.g. 2 or 10 for numbers; 26 for character strings
  - A minimum of $\log_b N$ digits are needed to represent N distinct key values,
    - If there are N unique keys, r is in $\Omega(\log N)$

- The asymptotic complexity of Radix Sort is $\Omega(N\log N)$.

- Radix Sort is stable, not "In-place"

Homework 5-3

- To Implement all sorting algorithms discussed. (The mission is not homework.)
- According to the Radix sorting algorithm for variable-length strings in textbook Figure 7.27, show the sorting process for the input "We, can, extend, either, version, of, radix, sort, to, work, with, variable, length, strings".

- Deadline: to be confirmed.