Bucket Sort

Counting Sort succeeds by constructing a much smaller set of *k* values in which to count the *n* elements in the set. Given a set of *n* elements, Bucket Sort constructs a set of *n* buckets into which the elements of the input set are partitioned; Bucket Sort thus reduces its processing costs at the expense of this extra space. If a hash function, hash(A_i), is provided that uniformly partitions the input set of *n* elements into these *n* buckets, then Bucket Sort as described in Figure 4-18 can sort, in the worst case, in O(*n*) time. You can use Bucket Sort if the following two properties hold:

Uniform distribution

The input data must be uniformly distributed for a given range. Based on this distribution, *n* buckets are created to evenly partition the input range.

Ordered hash function

v_j.

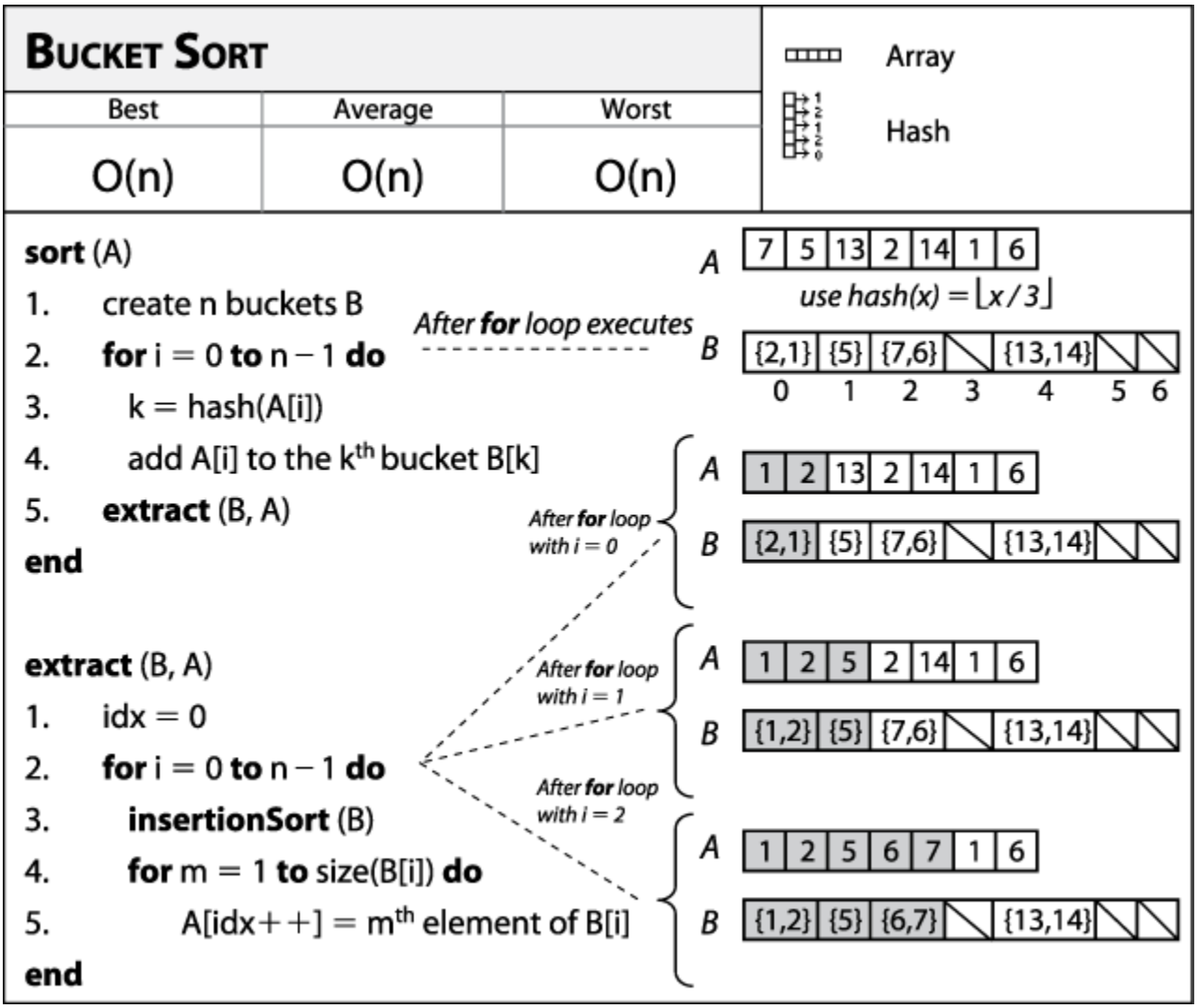

Figure 4-18. Bucket Sort fact sheet

Bucket Sort is not appropriate for sorting arbitrary strings, for example; however, it could be used to sort a set of uniformly distributed floating-point numbers in the range [0,1).

Once all elements to be sorted are inserted into the buckets, Bucket Sort extracts the values from left to right using Insertion Sort on the contents of each bucket. This orders the elements in each respective bucket as the values from the buckets are extracted from left to right to repopulate the original array.

partitioned using a fast hashing function.

Forces

If storage space is not important and the elements admit to an immediate total ordering, Bucket Sort can take advantage of this extra knowledge for impressive cost savings.

Solution

In the C implementation for Bucket Sort, shown in Example 4-11, each bucket stores a linked list of elements that were hashed to that bucket. The functions `numBuckets` and `hash` are provided externally, based upon the input set.

Example 4-11. Bucket Sort implementation in C

```
extern int hash(void *elt);
extern int numBuckets(int numElements);

/* linked list of elements in bucket. */
typedef struct entry {
  void          *element;
  struct entry  *next;
} ENTRY;

/* maintain count of entries in each bucket and pointer to its first entr
typedef struct {
  int        size;
  ENTRY      *head;
} BUCKET;

/* Allocation of buckets and the number of buckets allocated */
static BUCKET *buckets = 0;
static int num = 0;
```

```
  int i, low;
  int idx = 0;
  for (i = 0; i < num; i++) {
    ENTRY *ptr, *tmp;
    if (buckets[i].size == 0) continue;   /* empty bucket */

    ptr = buckets[i].head;
    if (buckets[i].size == 1) {
      ar[idx++] = ptr->element;
      free (ptr);
      buckets[i].size = 0;
      continue;
    }

    /* insertion sort where elements are drawn from linked list and
     * inserted into array. Linked lists are released. */
    low = idx;
    ar[idx++] = ptr->element;
    tmp = ptr;
    ptr = ptr->next;
    free (tmp);
```

```
while (ptr != NULL) {
     int i = idx-1;
     while (i >= low && cmp (ar[i], ptr->element) > 0) {
        ar[i+1] = ar[i];
        i--;
      }
      ar[i+1] = ptr->element;
      tmp = ptr;
      ptr = ptr->next;
      free(tmp);
      idx++;
    }
    buckets[i].size = 0;
  }
}
```

```
  int i;
  num = numBuckets(n);
  buckets = (BUCKET *) calloc (num, sizeof (BUCKET));
  for (i = 0; i < n; i++) {
    int k = hash(ar[i]);

    /** Insert each element and increment counts */
    ENTRY *e = (ENTRY *) calloc (1, sizeof (ENTRY));
    e->element = ar[i];
    if (buckets[k].head == NULL) {
      buckets[k].head = e;
    } else {
      e->next = buckets[k].head;
      buckets[k].head = e;
    }

    buckets[k].size++;
  }

  /* now read out and overwrite ar. */
  extract (buckets, cmp, ar, n);

  free (buckets);
}
```

For numbers drawn uniformly from [0,1), Example 4-12 contains sample implementations of the `hash` and `numBuckets` functions to use.

Example 4-12. hash and numBuckets functions for [0,1) range

```
static int num;

/** Number of buckets to use is the same as the number of elements. */
int numBuckets(int numElements) {
```

```
,

/**
 * Hash function to identify bucket number from element. Customized
 * to properly encode elements in order within the buckets. Range of
 * numbers is from [0,1), so we subdivide into buckets of size 1/num;
 */
int hash(double *d) {
  int bucket = num*(*d);
  return bucket;
}
```

The buckets could also be stored using fixed arrays that are reallocated when the buckets become full, but the linked list implementation is about 30-40% faster.

Analysis

In the `sortPointers` function of Example 4-11, each element in the input is inserted into its associated bucket based upon the provided `hash` function; this takes linear, or O(n), time. The elements in the buckets are not sorted, but because of the careful design of the `hash` function, we know that all elements in bucket b_i are smaller than the elements in bucket b_j, if $i<j$.

As the values are extracted from the buckets and written back into the input array, Insertion Sort is used when a bucket contains more than a single element. For Bucket Sort to exhibit O(n) behavior, we must guarantee that the total time to sort each of these buckets is also O(n). Let's define n_i to be the number of elements partitioned in bucket b_i. We can treat n_i as a random variable (using statistical theory). Now consider the expected value $E[n_i]$ of n_i. Each element in the input set has probability $p=1/n$ of being inserted into a given bucket because each of these elements is uniformly drawn from the range [0,1). Therefore,

more than one element; we need to be sure that no bucket has too many elements. Once again, we resort to statistical theory, which provides the following equation for random variables:

$$E[n_i^2] = Var[n_i] + E^2[n_i]$$

From this equation we can compute the expected value of n_i^2. This is critical because it is the factor that determines the cost of Insertion Sort, which runs in a worst case of $O(n^2)$. We compute $E[n_i^2]=(1-1/n)+1=(2-1/n)$, which shows that $E[n_i^2]$ is a constant. This means that when we sum up the costs of executing Insertion Sort on all n buckets, the expected performance cost remains $O(n)$.

Variations

In Hash Sort, each bucket reflects a unique hash code value returned by the hash function used on each element. Instead of creating n buckets, Hash Sort creates a suitably large number of buckets k into which the elements are partitioned; as k grows in size, the performance of Hash Sort improves. The key to Hash Sort is a hashing function `hash(e)` that returns an integer for each element e such that $hash(a_i) \leq hash(a_j)$ if $a_i \leq a_j$.

The hash function `hash(e)` defined in Example 4-13 operates over elements containing just lowercase letters. It converts the first three characters of the string into a value (in base 26), and so for the string "abcdefgh," its first three characters ("abc") are extracted and converted into the value 0*676+1*26+2=28. This string is thus inserted into the bucket labeled 28.

Example 4-13. hash and numBuckets functions for Hash Sort

```
/** Number of buckets to use. */
int numBuckets(int numElements) {
```

```
/**
 * Hash function to identify bucket number from element. Customized
 * to properly encode elements in order within the buckets.
 */
int hash(void *elt) {

  return (((char*)elt)[0] - 'a')*676 +
         (((char*)elt)[1] - 'a')*26 +
         (((char*)elt)[2] - 'a');
}
```

The performance of Hash Sort for various bucket sizes and input sets is shown in Table 4-5. We show comparable sorting times for Quicksort using the median-of-three approach for selecting the `pivotIndex`.

n	26 buckets	676 buckets	17,576 buckets	Quicksort
16	0.000007	0.000026	0.000353	0.000006
32	0.00001	0.000037	0.000401	0.000007
64	0.000015	0.000031	0.000466	0.000016
128	0.000025	0.000042	0.000613	0.000031
256	0.000051	0.000062	0.00062	0.000045
512	0.000108	0.000093	0.000683	0.000098
1,024	0.000337	0.000176	0.0011	0.000282
2,048	0.0011	0.000456	0.0013	0.000637
4,096	0.0038	0.0012	0.0018	0.0017

8,192	0.0116	0.0027	0.0033	0.0037
16,384	0.048	0.0077	0.0069	0.009
32,768	0.2004	0.0224	0.0162	0.0207
65,536	0.8783	0.0682	0.0351	0.0525
131,072	2.5426	0.1136	0.0515	0.1151

Note that with 17,576 buckets, Hash Sort outperforms Quicksort for n>8,192 items (and this trend continues with increasing n). However, with only 676 buckets, once n>32,768 (for an average of 48 elements per bucket), Hash Sort begins its inevitable slowdown with the accumulated cost of executing Insertion Sort on increasingly larger sets. Indeed, with only 26 buckets, once n>256, Hash Sort begins to quadruple its performance as the problem size doubles, showing how too few buckets leads to $O(n^2)$ performance.