10. Hashing

Oliver Diessel
odiessel@cse.unsw.edu.au

Last updated: 15:00 17 Sep 2008
File Operations

- Topics to be covered:
  - Dictionaries and searching
  - Hashing
    - Dense and sparse integer ranges
    - Character strings
Dictionaries

- One of the common activities performed on data is to search a set of values looking for the one that matches a specified key.
- There are many data structures and algorithms for tackling this problem.

- For example, in an array of structs storing student information, it might be necessary to search for a given student number in order to extract a set of subjects and marks for that student.
  - If the array is not sorted by student number, the searching process must be linear, or sequential: the search must start with the first object and check each one in turn ⇒ linear search in a list or array of $n$ items requires $O(n)$ time
  
  - Imagine looking in the phone book to find out who corresponds to the number 9876-5432.
The *dictionary* abstract data type

- *Abstract data types* so far encountered include the *stack* and the *queue*.

- A *dictionary* is a new type of abstract data type.
- This ADT supports operations to “insert a new item”, “find the item that matches this key”, and “delete this item”.
- Sometimes a “process all items in sorted order” operation is also required.

- Binary search trees, linked lists, and arrays all provide a way of supporting the dictionary operations.
- However, different implementation choices provide different trade-offs between the costs of the various operations.
Hashing

- *Hashing* is a further way of providing a dictionary structure.

- If the keys are dense intervals of integers, then an array can be used, and accessed using the key as an index.

- For example, if the keys are staff ID numbers in the range 1,000 to 9,999, then an array of 10,000 pointers to structures for storing staff data could be allocated and accessed directly by staff ID number
  - Efficient access: search, insert, delete all take constant time
  - Efficient storage: for a few thousand staff, allocating 10,000 pointers is not unreasonable or inefficient
Hashing

- *Hashing* is a further way of providing a dictionary structure.

- If the keys are dense intervals of integers, then an array can be used, and accessed using the key as an index.

- If the keys are not dense, or if they are not integers, a *hash function* can be used to generate integer values over a smaller range.
Example – sparse keys

- If staff ID numbers are eight-digit values in the range 10,000,000 to 99,999,999 and only a few thousand are in use at any time, then using an array with 100 million pointers is wasteful.
  - Could use an array of 10,000 pointers and index using the last 4 digits of the staff number

- Unless all staff numbers are unique in the last 4 digits, this approach results in collisions

- Create an array of 10,000 linked lists instead, with each list storing the set of objects sharing the last 4 digits
  - This is efficient if the lists are short, which can be guaranteed if the array size is larger than the number of staff and staff numbers are randomly distributed
Hashing

- Hashing is a further way of providing a dictionary structure.

- If the keys are dense intervals of integers, then an array can be used, and accessed using the key as an index.

- If the keys are not dense, or if they are not integers, a hash function can be used to generate integer values over a smaller range.

A hash function converts a value drawn from a large or indeterminate range into a seemingly random integer over a constrained range.

- These values can then be used to index an array. Each element in the array is a linked list of the items that hash to that integer value.
Example – sparse keys continued

- Using the last 4 digits of an 8-digit staff number to index into an array of 10,000 lists causes staff numbers 86-86-4007, 87-86-4007 and 88-86-4007 to collide on index 4007.

- A better, more random distribution of hash values can be produced when all digits of the staff number are used to form a hash value.

- For example, take the largest prime number less than 10,000 (9,973) and form the remainder of each staff number mod 9,973: 86-86-4007 is thereby hashed to 9150, 87-86-4007 to 1877 and 88-86-4007 to 4577.
Hashing strings

- Successful hashing of character strings – where success is measured according to the seeming randomness of the hash value for realistic sets of input data – is even more of a challenge.

- One approach is to accumulate the value of hashing the individual characters of the string.

- The following pair of functions illustrate how this can be done.
Initialising a hash function for strings

#define NVALUES 10

typedef struct {
    unsigned nvalues, *values, tabsize;
} hash_t;

hash_t *create_hash(unsigned seed, unsigned tabsize) {
    int i; hash_t *h;
    /* allocate the required memory space */
    h = malloc(sizeof(*h));
    assert(h != NULL);
    h->values = malloc(NVALUES*sizeof(*(h->values)));
    assert(h->values != NULL);
    h->nvalues = NVALUES;
    h->tabsize = tabsize;
    /* start the random number generator */
    srand(seed);
    /* then create a sequence of prime numbers from it */
    for (i=0; i<NVALUES; i++)
        h->values[i] = nextprime(tabsize + rand()%tabsize);
    return h;
}
Calculating a hash value for a string

typedef struct {
    unsigned nvalues, *values, tabsize;
} hash_t;

unsigned calculate_hash(hash_t *h, char *key) {
    int i, k=0;
    unsigned hval=0;
    /* first, involve every character in the string */
    for (i=0; key[i]!='$\0$'; i++) {
        k = i % h->nvalues;
        hval += h->values[k]*key[i];
    }
    /* then reduce into the desired range */
    return hval % h->tabsize;
}
Distribution of keys for strings

- When 45,000 strings are hashed into a table of size 10,000 using these functions, each table entry (list) should receive an average of 4.5 items, and the distribution of list lengths should be a smooth curve.
  - Approximately 1% of buckets were unused.
  - No bucket received more than 17 items.
A faster hash function for strings?

- The previous computation is costly; one may therefore be tempted to try a faster alternative:

  ```c
  unsigned calculate_hash(hash_t *h, char *key) {
    int i; unsigned hval=0;
    for (i=0; key[i]!='$\0$'; i++)
      hval = hval + key[i];
    return hval % h->tabsize;
  }
  ```

- But then everything goes wrong:
  - Roughly 80% of buckets go unused
  - The average bucket size rises to 71.5
  - The largest bucket contains 163 strings

- The distribution becomes hopelessly skewed because ASCII values range up to 127 and average words of length 8 then map into only about 1,000 hash values
Hash functions

- The behaviour of a hashing scheme is critically dependent on the quality of the hash function, and the ratio between the number of items $n$, and the size of the array.

- Do:
  - involve all digits or characters of the key in the hash function;
  - make the table size big enough to allow for values of $n$ larger than you currently anticipate; and
  - test your hash function, to be sure that it is distributing keys across the target integers.
Hash functions

Don’t:

- presume that the input data is initially random, (it probably won’t be);
- make the mistake of thinking “it will be alright” (if you are careless, hashing can be very expensive in terms of execution time).

A hash function should always be checked on realistic data, to ensure it is distributing input values randomly, and to verify that the majority of buckets are getting used.
Where to from here?

- **Reading**
  - Hashing (Moffat, Ch. 12)

- **Next**
  - Lectures for Weeks 9 – 12 given by Wayne Wobcke
  - Lectures will cover:
    - Graph data structures and algorithms
    - Algorithmic complexity
    - Recursion
    - Sorting