Lesson 4. Searching (2): Binary Search and Hash Method

I111E – Algorithms and Data Structures

Ryuhei Uehara & Giovanni Viglietta
`uehara@jaist.ac.jp` & `johnny@jaist.ac.jp`

JAIST – October 28, 2019

All material is available at
`www.jaist.ac.jp/~uehara/couse/2019/i111e`

I111E – Q&A time

Question:

In the m-block searching method, in the worst case we perform n/m comparisons to find the block that may contain the number, and then m comparisons to find the number within the block.
In the second phase, couldn't we spare 1 comparison, since we have already checked the last number in the block?
Shouldn't the upper bound be $\underline{n/m + m - 1}$ instead of $n/m + m$?

I111E – Q&A time

Question:

In the m-block searching method, in the worst case we perform n/m comparisons to find the block that may contain the number, and then m comparisons to find the number within the block.
In the second phase, couldn't we spare 1 comparison, since we have already checked the last number in the block?
Shouldn't the upper bound be $\underline{n/m + m - 1}$ instead of $n/m + m$?

Answer:

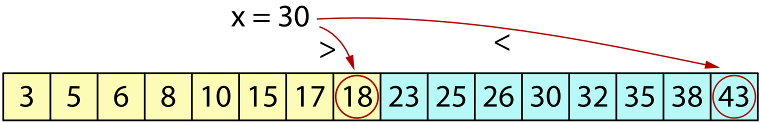
The observation is correct, and it can be an optimization of the m-block method. However, the m-block method itself does not implement this optimization, and therefore still performs $n/m + m$ comparisons in the worst case.
The optimization does indeed reduce the worst case by 1 comparison, but this becomes negligible asymptotically.
If we want to improve our searching method, we should first look for asymptotically better solutions (e.g., binary search :)).

Goals of today's lecture

- Learn binary search as a refinement of the m-block method
- Learn about hash functions and the hash method
- Familiarize with C structs, pointers, and memory allocation

Double 2-block method, revisited

Recall the double m-block search method with $m = 2$:



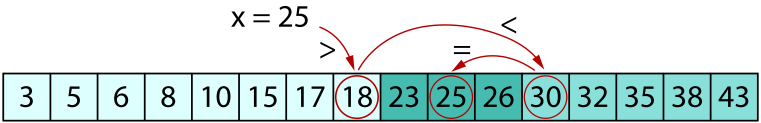
We check the middle and the last element of the array to know if x is in the first block or in the second block.

But do we really have to check the last element?

If x is greater than the middle element,
we already know it can only be in the second block!

Binary search

Idea of binary search: test only the middle element!



- If $s[\mathrm{mid}] = x$, return mid.
- If $s[\mathrm{mid}] > x$, search for x in the left half of the array.
- If $s[\mathrm{mid}] < x$, search for x in the right half of the array.

Repeat the above in the chosen interval

until x is found or the interval becomes empty.

Binary search

Implementation idea: two variables `left` and `right` mark the endpoints of the interval we are searching.

```
int binary_search(int s[], int n, int x) {
    int left = 0;
    int right = n - 1;
    do {
        int mid = (left + right) / 2;
        if (s[mid] == x) return mid;
        if (s[mid] > x) right = mid - 1;
        else left = mid + 1;
    } while (left <= right);
    return -1;
}
```

Binary search

Implementation idea: two variables `left` and `right` mark the endpoints of the interval we are searching.

```
int binary_search(int s[], int n, int x) {
    int left = 0;
    int right = n - 1;
    do {
        int mid = (left + right) / 2;
        if (s[mid] == x) return mid;
        if (s[mid] > x) right = mid - 1;
        else left = mid + 1;
    } while (left <= right);
    return -1;
}
```

At each repetition of the loop, the interval's size reduces by half. Hence the number of repetitions is $O(\log n)$.

Each repetition performs a constant number of operations. So, the total running time is $O(\log n)$.

Optimality of binary search

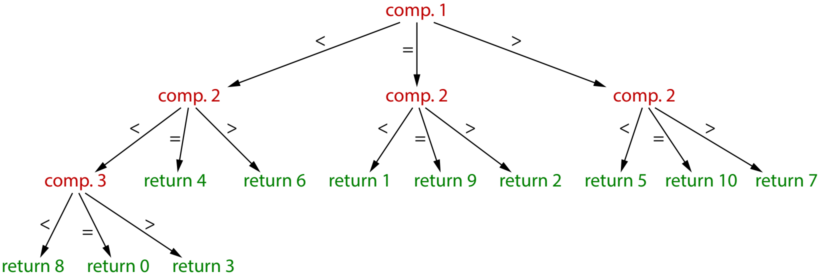
Is $O(\log n)$ an optimal running time for searching an array?

Suppose that all we can do are comparisons between integers.

Each comparison gives one of 3 possible outcomes: $<$, $=$, $>$.

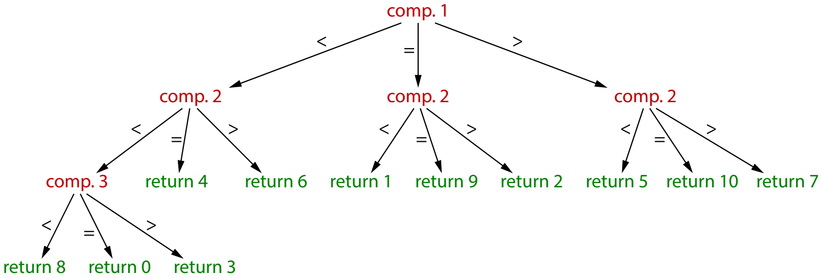
Depending on the outcome, we perform more comparisons, etc.

The whole search algorithm can be represented as a ternary tree:



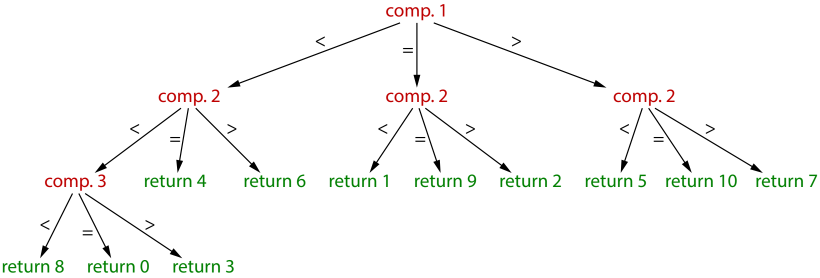
Note that, depending on the input, the search algorithm may output any of the n elements of the array.

Optimality of binary search



This means that the search tree must have at least <u>n different leaves</u>, corresponding to all possible outputs.

Optimality of binary search



This means that the search tree must have at least n different leaves, corresponding to all possible outputs.

Lemma: a ternary tree of height h has at most 3^h leaves.

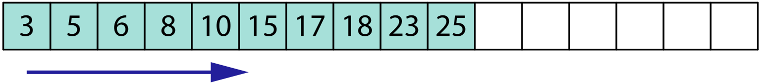
It follows that $3^h \geq n$, which implies that $h \geq \log_3 n$.

But the height of the search tree h is also the number of comparisons performed by the search algorithm in the worst case!

So, the number of comparisons must be at least $\log_3 n = O(\log n)$.

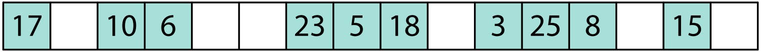
Hashing

So far, we have considered arrays where data is packed in order:



3	5	6	8	10	15	17	18	23	25						

Next we are going to assume that data is distributed differently:

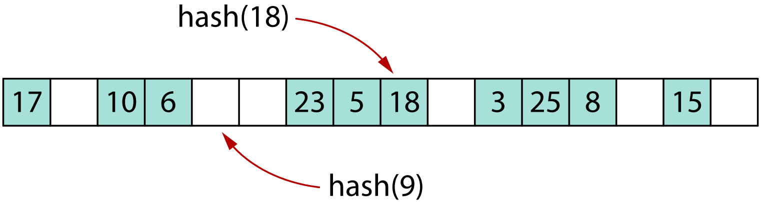


17		10	6			23	5	18		3	25	8		15	

We assume there is a hash function hash
that maps a value x to an array position $i = \mathrm{hash}(x)$.

Hash method

The hash method is based on the assumption that the value x is stored in $s[\mathrm{hash}(x)]$.

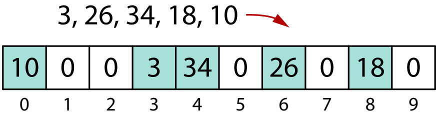


So, all we have to do to search for x is compute $\mathrm{hash}(x)$.

New data can also be inserted by computing $\mathrm{hash}(x)$.

Hash method

A typical hash function is $\mathrm{hash}(x) = x \bmod n$, where n is the size of the array:



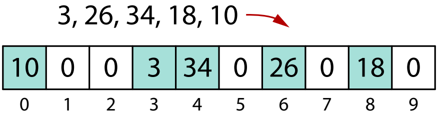
3, 26, 34, 18, 10 →

10	0	0	3	34	0	26	0	18	0
0	1	2	3	4	5	6	7	8	9

Hash method

A typical hash function is $\mathrm{hash}(x) = x \bmod n$,

where n is the size of the array:

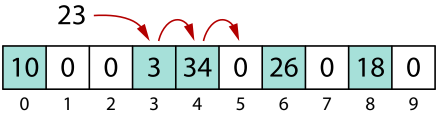


3, 26, 34, 18, 10

0	1	2	3	4	5	6	7	8	9
10	0	0	3	34	0	26	0	18	0

What if we want to insert x, but $s[\mathrm{hash}(x)]$ is already occupied?

This is called a collision.



23

0	1	2	3	4	5	6	7	8	9
10	0	0	3	34	0	26	0	18	0

In this case, we store x in the first empty cell after $s[\mathrm{hash}(x)]$.

Hash method: insertion

Here is an implementation of our insertion algorithm:

```
void hash_insert(int s[], int n, int x) {
   int i = x % n;
   while (s[i] != 0) {
      if (s[i] == x) return;
      i = (i + 1) % n;
   }
   s[i] = x;
}
```

Hash method: insertion

Here is an implementation of our insertion algorithm:

```
void hash_insert(int s[], int n, int x) {
    int i = x % n;
    while (s[i] != 0) {
        if (s[i] == x) return;
        i = (i + 1) % n;
    }
    s[i] = x;
}
```

The running time depends on whether there is a collision or not:

- If there are no collisions, the running time is constant: $O(1)$.
- If there are collisions, we may need to scan the whole array before finding a free cell: in the worst case, this takes $O(n)$.

Hash method: search

The search algorithm is very similar to the insertion one:

```
int hash_search(int s[], int n, int x) {
   int i = x % n;
   while (s[i] != 0) {
      if (s[i] == x) return i;
      i = (i + 1) % n;
   }
   return -1;
}
```

Once again, the running time depends on collisions,

and it may range from $O(1)$ to $O(n)$.

Hash method: performance

If an array of size n contains m occupied cells,

we define the occupation ratio (or load factor) as $\alpha = m/n$.

The expected number of array accesses of the hash method is:

- $\approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ if the value is found,
- $\approx \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$ if the value is not found.

Hash method: performance

If an array of size n contains m occupied cells,

we define the occupation ratio (or load factor) as $\alpha = m/n$.

The expected number of array accesses of the hash method is:

- $\approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ if the value is found,
- $\approx \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$ if the value is not found.

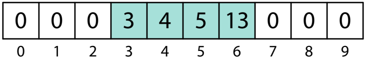
If the array is large compared to the number of stored data, the search time is likely to be constant, but a lot of memory is wasted.

On the other hand, if all cells are occupied, no memory is wasted, but the hash method may take a long time, or even loop forever!

The number of collisions usually depends on the distribution of data and on the hash function used. Good hash functions may reduce collisions, but may be hard to compute: not just $O(1)$.

Hash method: deletion

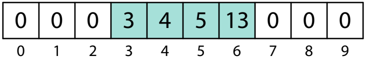
Unfortunately, deleting data from a hash table is not a simple task. Suppose that $n = 10$, the hash function is $\mathrm{hash}(x) = x \bmod 10$, and we have inserted the values 3, 4, 5, 13, in this order:



0	0	0	3	4	5	13	0	0	0
0	1	2	3	4	5	6	7	8	9

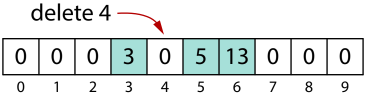
Hash method: deletion

Unfortunately, deleting data from a hash table is not a simple task. Suppose that $n = 10$, the hash function is $\mathrm{hash}(x) = x \bmod 10$, and we have inserted the values 3, 4, 5, 13, in this order:



0	1	2	3	4	5	6	7	8	9
0	0	0	3	4	5	13	0	0	0

Let us try to delete 4 in a naive way:

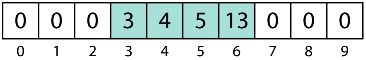


delete 4

0	1	2	3	4	5	6	7	8	9
0	0	0	3	0	5	13	0	0	0

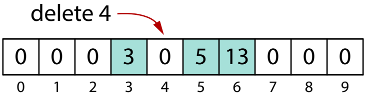
Hash method: deletion

Unfortunately, deleting data from a hash table is not a simple task. Suppose that $n = 10$, the hash function is $\mathrm{hash}(x) = x \bmod 10$, and we have inserted the values 3, 4, 5, 13, in this order:



0	0	0	3	4	5	13	0	0	0
0	1	2	3	4	5	6	7	8	9

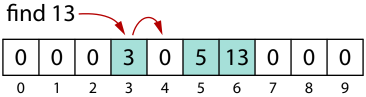
Let us try to delete 4 in a naive way:



delete 4

0	0	0	3	0	5	13	0	0	0
0	1	2	3	4	5	6	7	8	9

Now, if we search for 13, we don't find it!



find 13

0	0	0	3	0	5	13	0	0	0
0	1	2	3	4	5	6	7	8	9

To delete an entry properly, we would have to scan the whole array and perform some complex substitutions:
this type of hash method is quick, but data is hard to maintain!

Structuring data in C

In C, we can group different variables into a single structure, and thus define a new type of variable:

```
typedef struct {
    int var1;
    int var2;
    int var3;
} my_structure;
```

Now we can use `my_structure` as a variable type, and access the individual fields of a `my_structure` variable:

```
my_structure s;
s.var1 = 3;
s.var2 = 5;
s.var3 = 2;
printf("%d", s.var1 + s.var2 + s.var3);
```

The fields of a structure can be simple variables, arrays, or even other structures, arrays of structures, etc.

Allocating memory in C

There is a C function to request some memory to the operating system: `malloc`. This function takes as input the amount of memory we want, and outputs a pointer, i.e., the *address* of the memory allocated by the operating system.

We can also declare a variable of pointer type (just add “*” after the type), and use it to store the pointer returned by `malloc`:

```
int* x = malloc(sizeof(int));
```

If we use `malloc` to allocate a structure,
we can access its fields with “->” instead of “.”:

```
my_structure* s = malloc(sizeof(my_structure));
s -> var1 = 3;
s -> var2 = s -> var1 + s -> var1;
```

When we are done using a variable allocated with `malloc`, we can return the memory to the operating system using `free`:

```
free(s);
```

Self-referential structures in C

We can also declare a structure type containing a pointer to the same structure type as a field:

```
typedef struct node {
    int data;
    struct node* other;
} node;
```

Now, each variable of type `node` can be "attached" to some other variable of type `node`. (To denote a pointer that doesn't point to anything, we use the value `NULL`.)

```
node* n1 = malloc(sizeof(node));
node* n2 = malloc(sizeof(node));
n1 -> data = 4;
n1 -> other = n2;
n2 -> data = 9;
n2 -> other = NULL;
```