CH08-320201

Algorithms and Data Structures

ADS

Lecture 17

Dr. Kinga Lipskoch

Spring 2019

Choosing a Hash Function (1)

- What makes a good hash function?
  - The goal for creating a hash function is to distribute the keys as uniformly as possible to the slots.
- Division method
  - Define hashing function $h(k) = k \bmod m$.
  - Deficiency: Do not pick an m that has a small divisor d, as a prevalence of keys with the same modulo d can negatively effect uniformity.
  - Example: if m is a power of 2, the hash function only depends on a few bits: If $k = 1011000111011010$ and $m = 2^6$, then $h(k) = 011010$.

Choosing a Hash Function (2)

- Division method (continue)
  - Common choice: Pick m to be a prime not too close to a power of 2 or 10 and not otherwise prominently used in computing environments.
  - Example: $n = 2000$; we are ok with average 3 elements in our collision chain $\Rightarrow m = 701$ (a prime number close to $2000/3$), $h(k) = k \bmod 701$.

Choosing a Hash Function (3)

- Multiplication method
  - On advantage of the multiplication method is that the value of m is not critical
  - Knuth suggests that $A \approx (\sqrt{5} - 1)/2$ works well
  - Assume all keys are integers, $m = 2^r$, and the computer uses w-bit words.
  - Define hash function $h(k) = (A \cdot k \bmod 2^w) >> (w - r)$, where "$>>$" is the right bit-shift operator and A is an odd integer with $2^{w-1} < A < 2^w$.
  - Example: $m = 2^3 = 8$ and $w = 7$.

$$\begin{array}{rl} 1011001 & = A \\ \times \qquad\qquad 1101011 & = k \\ \hline 1001010\underbrace{011}_{h(k)}0011 & \end{array}$$

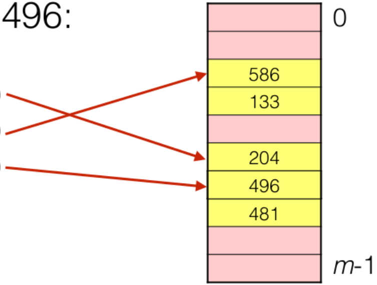
Resolving Collisions by Open Addressing

- No additional storage is used.
- Only store one element per slot.
- Insertion probes the table systematically until an empty slot is found.
- The hash function depends on the key and the probe number, i.e., $h : U \times \{0, 1, ..., m-1\} \rightarrow \{0, 1, ..., m-1\}$.
- The probe sequence $< h(k, 0), h(k, 1), ..., h(k, m-1) >$ should be a permutation of $\{0, 1, ..., m-1\}$.

Insert Example

- Insert key k = 496:

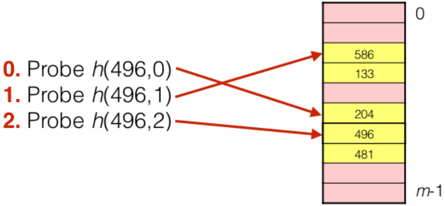
0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$




```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

Search Example

```
HASH-SEARCH(T, k)
  i = 0
  repeat
      j = h(k, i)
      if T[j] == k
          return j
      i = i + 1
  until T[j] == NIL or i == m
  return NIL
```



- Search key $k = 496$
  - Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot (or made it all the way through the list)
  - Search times no longer depend on load factor α
- What about delete?
  - Have a special node type: DELETED
  - Chaining more commonly used when keys must also be deleted

Probing Strategies (1)

Linear probing:

- Given an ordinary hash function $h'(k)$, linear probing uses the hash function $h(k, i) = (h'(k) + i) \bmod m$.
- This is a simple computation.
- However, it may suffer from primary clustering, where long runs of occupied slots build up and tend to get longer.
  - empty slot preceded by i full slots gets filled next with probability $(i+1)/m$

Probing Strategies (2)

Quadratic probing:

- Quadratic probing uses the hash function $h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$.
- Offset by amount that depends on quadratic manner, works much better than linear probing
- But, it may still suffer from secondary clustering: If two keys have initially the same value, then they also have the same probe sequence
- In addition c_1, c_2, and m need to be constrained to make full use of the hash table

Probing Strategies (3)

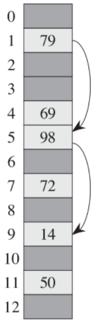
Double hashing:

- Given two ordinary hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$.
- The initial probe goes to position $T[h_1(k)]$; successive probe positions are offset by $h_2(k) \rightarrow$ the initial probe position, the offset, or both, may vary
- This method generates excellent results, if $h_2(k)$ is relatively prime to the hash-table size m,

Probing Strategies (4)

Double hashing (continue):

- e.g., by making m a power of 2 and design $h_2(k)$ to only produce odd numbers.
- or let m be prime and design h_2 such that it always returns a positive integer less than m, e.g. let m' be slightly less than m:
  $h_1(k) = k \bmod m$
  $h_2(k) = 1 + (k \bmod m')$



$h_1(k) = k \bmod 13$
$h_2(k) = 1 + (k \bmod 11)$
—> $k=14$; $h_1(k)=1$, $h_2(k)=4$
—> $k=27$; $h_1(k)=1$, $h_2(k)=6$

Analysis of Open Addressing (1)

Theorem:

- Assume uniform hashing, i.e., each key is likely to have any one of the $m!$ permutations as its probe sequence.
- Given an open-addressed hash table with load factor $\alpha = n/m < 1$.
- The expected number of probes in an unsuccessful search is, at most $\dfrac{1}{1-\alpha}$.

Analysis of Open Addressing (2)

Proof:

- At least, one probe is always necessary.
- With probability n/m, the first probe hits an occupied slot, i.e., a second probe is necessary.
- With probability $(n-1)/(m-1)$, the second probe hits an occupied slot, i.e., a third probe is necessary.
- With probability $(n-2)/(m-2)$, the third probe hits an occupied slot, i.e., a fourth probe is necessary.
- ...

Analysis of Open Addressing (3)

Given that $\dfrac{n-i}{m-i} < \dfrac{n}{m} = \alpha$ for $i = 1, 2, \ldots, n$.

$$
\begin{aligned}
&1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\left(1+\frac{1}{m-n+1}\right)\cdots\right)\right)\right)\\
&\le 1+\alpha(1+\alpha(1+\alpha(\cdots(1+\alpha)\cdots)))\\
&\le 1+\alpha+\alpha^2+\alpha^3+\cdots\\
&=\sum_{i=0}^{\infty}\alpha^i\\
&=\frac{1}{1-\alpha}.
\end{aligned}
$$

Analysis of Open Addressing (4)

- The successful search takes less number of probes $\left[\text{expected number is at most } \dfrac{1}{\alpha}\ln\dfrac{1}{1-\alpha}\right]$.
- We conclude that if α is constant, then accessing an open-addressed hash table takes constant time.
- For example, if the table is half full, the expected number of probes is $1/(1-0.5)=2$.
- Or, if the table is 90% full, the expected number of probes is $1/(1-0.9)=10$.

Summary

- Dynamic sets with queries and modifying operations.
- Array: Random access, search in $O(\lg n)$, but modifying operations $O(n)$.
- Stack: LIFO only. Operations in $O(1)$.
- Queue: FIFO only. Operations in $O(1)$.
- Linked list: Modifying operations in $O(1)$, but search $O(n)$.
- BST: All operations in $O(h)$.
- Red-black trees: All operations in $O(\lg n)$.
- Heap: All operations in $O(\lg n)$.
- Hash tables: Operations in $O(1)$, but additional storage space.

Design Concepts

- We have been looking into different algorithms and, in particular, emphasized one design concept, namely, the Divide & Conquer strategy, which was based on recursions and whose analysis was given by recurrences.
- Now, we are going to look into further design concepts.

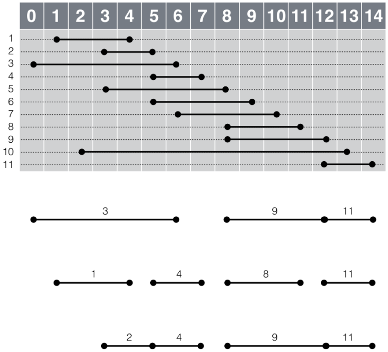
Activity-Selection Problem (1)

- Suppose we have a set $S = \{a_1, a_2, ..., a_n\}$ of n activities.
- The activities wish to use a resource, which can only be used by one activity at a time.
- Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$.
- Two activities a_i and a_j are compatible, if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint.
- The activity-selection problem is to select a maximum-size subset of mutually compatible activities.

Activity-Selection Problem (2)

- Example:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14



- $\{a_3, a_9, a_{11}\}$ is a subset of mutually compatible activities.
- $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities.
- $\{a_2, a_4, a_9, a_{11}\}$ is another largest subset of mutually compatible activities.

Sorting

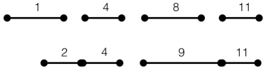
- We can apply a sorting algorithm to the finish times, which operates in $O(n \lg n)$ time.
- Then, we can assume that the activities are sorted, i.e., $f_1 \leq f_2 \leq ... \leq f_n$.

Greedy Algorithm

- A greedy algorithm always makes the choice that looks best at the moment.
- I.e., it makes a locally optimal choice in the hope that it will lead to a globally optimal solution.

Greedy Approach (1)

- After sorting, a_1 has the earliest finish time f_1.
- A greedy approach starts with taking a_1 as a locally optimal choice.
- Lemma:
  The greedy choice of picking a_1 as first choice is optimal.
- Proof:
  - Suppose A is a globally optimal solution for set S.
  - Let $a_k \in A$ be the activity with earliest finish time f_k in A.
  - If $k = 1$, then $a_1 \in A$ and we are done.
  - If $k > 1$, then we can replace A by $(A \setminus \{a_k\}) \cup \{a_1\}$.
  - Since $f_1 \leq f_k$, this is still an optimal solution.
  - Hence, we can always start with a_1.



Greedy Approach (2)

- After the first step, we consider the subproblem $S' = \{a_i \in S : s_i \geq f_1\}$.
- We apply the same greedy strategy.
- Lemma:
  $A \setminus \{a_1\}$ is the optimal solution for S'.
- Proof:
  - Let B be a solution for S' that is larger than $A \setminus \{a_1\}$.
  - Then, $B \cup \{a_1\}$ would be solution for S that is larger than A.
  - Contradiction.

Greedy Approach (3)

Using the two lemmata we can prove by induction that the greedy approach delivers the globally optimal solution.

Greedy Algorithm

```
Greedy-Selector(S)
  // Assume S = {a[1], ..., a[n]}
  // with activities sorted by f[i].
  A := {a[1]}
  j := 1
  for i := 2 to n do
    if s[i] >= f[j]
      then A := A union {a[i]}
           j := i
  return A
```

Greedy Algorithm (Recursive)

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

$m = k + 1$
while $m \leq n$ and $s[m] < f[k]$ // find the first activity in S_k to finish
  $m = m + 1$
if $m \leq n$
  return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return $\emptyset$

RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.