Biostatistics 615/815 Lecture 8: Hash Tables, and Dynamic Programming

Hyun Min Kang

February 1st, 2011

Recap: Elementary data structures

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Insert</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>SortedArray</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Tree</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>Hash</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

- Array or list is simple and fast enough for small-sized data
- Tree is easier to scale up to moderate to large-sized data
- Hash is the most robust for very large datasets

Recap: Example of a linked list

(a) L.head

(b) L.head

(c) L.head

815 projects
- Instructor sent out E-mails to individually today morning

Homework #2
- For problem 3, assume that all the input values are unique
- Include the class definition into myTree.h and myTreeNode.h (do not make .cpp file)
- The homework .tex file containing the source code is uploaded in the class web page
Recap: An example binary search tree

- Pointers to left and right children (NIL if absent)
- Pointers to its parent can be omitted.

Today

Data structure
- Hash table

Dynamic programming
- Divide and conquer vs dynamic programming

Containers for single-valued objects - last lectures
- \( \text{INSERT}(T, x) \) - Insert \( x \) to the container.
- \( \text{SEARCH}(T, x) \) - Returns the location/index/existence of \( x \).
- \( \text{REMOVE}(T, x) \) - Delete \( x \) from the container if exists
- STL examples include std::vector, std::list, std::deque, std::set, and std::multiset.

Containers for (key,value) pairs - this lecture
- \( \text{INSERT}(T, x) \) - Insert \( (x.key, x.value) \) to the container.
- \( \text{SEARCH}(T, k) \) - Returns the value associated with key \( k \).
- \( \text{REMOVE}(T, x) \) - Delete element \( x \) from the container if exists
- Examples include std::map, std::multimap, and _gnu_cxx::hash_map

Individually compile and link - Does NOT work with template
- Include the content of your .cpp files into .h
- For example, Main.cpp includes myArray.h

user@host:~> g++ -o myArrayTest Main.cpp
rm *.o myArrayTest

Or create a Makefile and just type 'make'

all: myArrayTest  # binary name is myArrayTest
myArrayTest: Main.cpp  # link two object files to build binary
g++ -o myArrayTest Main.cpp  # must start with a tab

clean:
rm *.o myArrayTest
Direct address tables

An example (key, value) container
- \( U = \{0, 1, \cdots, N - 1\} \) is possible values of keys (\( N \) is not huge)
- No two elements have the same key

Direct address table: a constant-time container
Let \( T[0, \cdots, N - 1] \) be an array space that can contain \( N \) objects
- \( \text{INSERT}(T, x) : T[x.key] = x \)
- \( \text{SEARCH}(T, k) : \text{return } T[k] \)
- \( \text{REMOVE}(T, x) : T[x.key] = \text{NIL} \)

Hash Tables

Key features
- \( O(1) \) complexity for \( \text{INSERT}, \text{SEARCH}, \) and \( \text{REMOVE} \)
- Requires large memory space than the actual content for maintaining good performance
- But uses much smaller memory than direct-address tables

Key components
- Hash function
  - \( h(x.key) \) mapping key onto smaller ‘addressable’ space \( H \)
  - Total required memory is the possible number of hash values
  - Good hash function minimize the possibility of key collisions
- Collision-resolution strategy, when \( h(k_1) = h(k_2) \).

Analysis of direct address tables

Time complexity
- Requires a single memory access for each operation
- \( O(1) \) - constant time complexity

Memory requirement
- Requires to pre-allocate memory space for any possible input value
- \( 2^{32} = 4GB \times (\text{size of data}) \) for 4 bytes (32 bit) key
- \( 2^{64} = 18EB (1.8 \times 10^7 \text{TB}) \times (\text{size of data}) \) for 8 bytes (64 bit) key
- An infinite amount of memory space needed for storing a set of arbitrary-length strings (or exponential to the length of the string)

Chained hash: A simple example

A good hash function
- Assume that we have a good hash function \( h(x.key) \) that ‘fairly uniformly’ distribute key values to \( H \)
- What makes a good hash function will be discussed later today.

A ChainedHash
- Each possible hash key contains a linked list
- Each linked list is originally empty
- An input \((k, v)\) pair is appended to the linked list when inserted
- \( O(1) \) time complexity is guaranteed when no collision occurs
- When collision occurs, the time complexity is proportional to size of linked list associated with \( h(x.key) \).
Illustration of ChainedHash

Flow of data in ChainedHash

Analysis of hashing with chaining

Assumptions

- Simple uniform hashing
  - $\Pr(h(k_1) = h(k_2)) = 1/m$ input key pairs $k_1$ and $k_2$.
- $n$ is the number of elements stores
- Load factor $\alpha = n/m$.

Expected time complexity for SEARCH

- $X_{ij} \in \{0, 1\}$ a random variable of key collision between $x_i$ and $x_j$.
- $E[X_{ij}] = 1/m$.

$$T(n) = \frac{1}{n} E \left[ \sum_{i=1}^{n} \left( 1 + \sum_{j=i+1}^{n} X_{ij} \right) \right] = \Theta(1 + \alpha)$$

Simplified algorithms on ChainedHash

INITIALIZE($T$)

- Allocate an array of list of size $m$ as the number of possible key values

INSERT($T, x$)

- Insert $x$ at the head of list $T[h(x.key)]$.

SEARCH($T, k$)

- Search for an element with key $k$ in list $T[h(k)]$.

REMOVE($T, x$)

- Delete $x$ from the list $T[h(x.key)]$.

Interesting properties (under uniform hash)

Probability of an empty slot

$$Pr(k_1 \neq k, k_2 \neq k, \cdots, k_n \neq k) = \left(1 - \frac{1}{m}\right)^n \approx e^{-\alpha}$$

Birthday paradox: expected # of elements before the first collision

$$Q(H) \approx \sqrt{\frac{\pi}{2}} \frac{1}{m}$$

Coupon collector problem: expect # of elements to fill every slot

$$\sum_{i=1}^{m} \frac{m}{i} \approx m(ln m + 0.577)$$
Hash functions

Making a good hash functions

- A hash function \( h(k) \) is a deterministic function from \( k \in K \) onto \( h(k) \in H \).
- A good hash function distributes map the keys to hash values as uniform as possible.
- The uniformity of hash function should not be affected by the pattern of input sequences.

Example hash functions

- \( k \in [0, 1) \), \( h(k) = \lfloor km \rfloor \)
- \( k \in \mathbb{N}, h(k) = k \mod m \)

Example of good hash functions

For integers

- Make the hash size \( m \) to be a large prime.
  - \( h(k) = k \mod m \)

For floating point values \( k \in [0, 1) \)

- Make the hash size \( m \) to be a large prime.
  - \( h(k) = \lfloor k \times N \rfloor \mod m \) for a large number \( N \).

For strings

- Pretend the string is a number with numeral system of \( |\Sigma| \), where \( \Sigma \) is the set of possible characters.
  - Apply the same hash function for integers.

'Good' and 'bad' hash functions

An example: \( h(k) = \lfloor km \rfloor \)

- When the input is uniformly distributed
  - \( h(k) \) is uniformly distributed between 0 and \( m - 1 \).
  - \( h(k) \) is a good hash function.
- When the input is skewed: \( \Pr(k < 1/m) = 0.9 \)
  - More than 80% of input key pairs will have collisions.
  - \( h(k) \) is a bad hash function.
  - Time complexity is close to a single linked list.

Good hash functions

- 'Goodness' of a hash function can be dependent on the data.
- If it is hard to create adversary inputs to make the hash function 'bad', it is generally a good hash function.

Open Addressing

Chained Hash - Pros and Cons

- Easy to understand.
- Behavior at collision is easy to track.
- Every slot maintains pointer - extra memory consumption.
- Inefficient to dereference pointers for each access.
- Larger and unpredictable memory consumption.

Open Addressing

- Store all the elements within an array.
- Resolve conflicts based on predefined probing rule.
- Avoid using pointers - faster and more memory efficient.
- Implementation of REMOVE can be very complicated.
Probing in open hash

**Modified hash functions**

- \( h : K \times H \rightarrow H \)
- For every \( k \in K \), the probe sequence \(< h(k, 0), h(k, 1), \ldots, h(k, m - 1) >\) must be a permutation of \(< 0, 1, \ldots, m - 1 >\).

---

**Algorithm OPENHASHINSERT**

**Data:** \( T : \text{hash} \), \( k : \text{key value to insert} \)

**Result:** \( k \) is inserted to \( T \)

for \( i = 0 \) to \( m - 1 \) do
  \( j = h(k, i) \)
  if \( T[j] == \text{NIL} \) then
    \( T[j] = k \)
  end
end
error "hash table overflow";

---

**Algorithm OPENHASHSEARCH**

**Data:** \( T : \text{hash} \), \( k : \text{key value to search} \)

**Result:** Return \( T[k] \) if exist, otherwise return \( \text{NIL} \)

for \( i = 0 \) to \( m - 1 \) do
  \( j = h(k, i) \)
  if \( T[j] == k \) then
    return \( j \)
  end
  else if \( T[j] == \text{NIL} \) then
    return \( \text{NIL} \)
  end
end
return \( \text{NIL} \);

---

**Strategies for Probing**

**Linear Probing**

- \( h(k, i) = (h'(k) + i) \mod m \)
- Easy to implement
- Suffer from primary clustering, increasing the average search time

**Quadratic Probing**

- \( h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m \)
- Better than linear probing
- Secondary clustering: \( h(k_1, 0) = h(k_2, 0) \) implies \( h(k_1, i) = h(k_2, i) \)
Strategies for Probing

Double Hashing
- \( h(k, i) = (h_1(k) + ih_2(k)) \mod m \)
- The probe sequence depends in two ways upon \( k \).
- For example, \( h_1(k) = k \mod m \), \( h_2(k) = 1 + (k \mod m') \)
- Avoid clustering problem
- Performance close to ideal scheme of uniform hashing.

Hash tables: summary
- Linear-time performance container with larger storage
- Key components
  - Hash function
  - Conflict-resolution strategy
- Chained hash
  - Linked list for every possible key values
  - Large memory consumption + deferencing overhead
- Open Addressing
  - Probing strategy is important
  - Double hashing is close to ideal hashing

When are binary search trees better than hash tables?
- When the memory efficiency is more important than the search efficiency
- When many input key values are not unique
- When querying by ranges or trying to find closest value.

Recap: Divide and conquer algorithms
- Good examples of divide and conquer algorithms
  - TowerOfHanoi
  - MergeSort
  - QuickSort
  - BinarySearchTree algorithms

These algorithms divide a problem into smaller and disjoint subproblems until they become trivial.
A divide-and-conquer algorithm for Fibonacci numbers

Fibonacci numbers

\[ F_n = \begin{cases} 
F_{n-1} + F_{n-2} & n > 1 \\
1 & n = 1 \\
0 & n = 0 
\end{cases} \]

A recursive implementation of Fibonacci numbers

```c
int fibonacci(int n) {
    if (n < 2) return n;
    else return fibonacci(n-1)+fibonacci(n-2);
}
```

Performance of recursive Fibonacci

Computational time

- 4.4 seconds for calculating \( F_{40} \)
- 49 seconds for calculating \( F_{45} \)
- \( \infty \) seconds for calculating \( F_{100} \)

Time complexity of redundant Fibonacci

\[
T(n) = T(n-1) + T(n-2)
\]

\[
\begin{align*}
T(1) &= 1 \\
T(0) &= 1 \\
T(n) &= F_{n+1}
\end{align*}
\]

The time complexity is exponential
A non-redundant **FIBONACCI**

```c
int fibonacci(int n) {
    int* fibs = new int[n+1];
    fibs[0] = 0;
    fibs[1] = 1;
    for(int i=2; i <= n; ++i) {
        fibs[i] = fibs[i-1]+fibs[i-2];
    }
    int ret = fibs[n];
    delete [] fibs;
    return ret;
}
```

Key idea in non-redundant **FIBONACCI**

- Each \( F_n \) will be reused to calculate \( F_{n+1} \) and \( F_{n+2} \)
- Store \( F_n \) into an array so that we don’t have to recalculate it

A recursive, but non-redundant **FIBONACCI**

```c
int fibonacci(int* fibs, int n) {
    if ( fibs[n] > 0 ) {
        return fibs[n]; // reuse stored solution if available
    }
    else if ( n < 2 ) {
        return n; // terminal condition
    }
    fibs[n] = fibonacci(n-1) + fibonacci(n-2); // store the solution once computed
    return fibs[n];
}
```

**Summary**

**Today**
- Hashing
- Dynamic programming

**Next Lecture**
- More on dynamic programming
- Graph algorithms

**Reading materials**
- CLRS Chapter 15