CS 261 – Data Structures

Hash Tables

Open Address Hashing
Midterm Exam 2
Midterm Exam 2

= 17%
Midterm Exam 2

= 26\%
Midterm Exam 2 – Problem 2.3

= 340\%
Total Scores
Total Scores

"genius" ≈ 15%
Total Scores

grade A
≈ 15%
Total Scores

> 55 => A
currently 31 students have an A
#include <stdio.h>
FILE *filePtr;
char filename[100];

filePtr = fopen(filename, "w");

if (filePtr == NULL)
    printf("Cannot open %s\n", filename);

fprintf(filePtr, "%d\t%s\n", task.priority, task.description);

fclose(filePtr);
#include <stdio.h>
FILE *filePtr;
char filename[100];
int priority;

filePtr = fopen(filename, "r");

if (filePointer == NULL)
    printf("Cannot open %s\n", filename);

while(fscanf(filePtr,"%d\t",&priority) != EOF)
    { ... }

fclose(filePtr);
#include <stdio.h>
FILE *filePtr;
char filename[100];
char desc[TASK_DESC_SIZE];

while(fscanf(filePtr,"%d\t", &priority) != EOF) {
    ... 
    fgets(desc, sizeof(desc), filePtr);
}
fclose(filePtr);
ADT Dictionaries

computer |kəmˈpyoʊtər|
noun

• an electronic device for storing and processing data...

• a person who makes calculations, esp. with a calculating machine.
Dictionaries

**computer** |kəmˈpjuːtər|

noun

• an electronic device for storing and processing data...

• a person who makes calculations, esp. with a calculating machine.
Dictionaries

**computer** |ˈkəmˌpyoʊtər|

noun

• an electronic device for storing and processing data...

• a person who makes calculations, esp. with a calculating machine.

**value**
How to implement dictionaries?
Hash Tables

Similar to dynamic arrays except:

1. Elements can be indexed by their keys whose type may differ from integer

2. In general, a single position may hold more than one element
Computing a Hash Table Index: 2 Steps

1. Transform the key to an integer
   • by using the hash function

2. Map the resulting integer to a valid hash table index
   • by using the remainder of dividing the integer with the table size
Example

Say, we’re storing names:

Angie
Joe
Abigail
Linda
Mark
Max
Robert
John

<table>
<thead>
<tr>
<th></th>
<th>Angie, Robert</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Linda</td>
</tr>
<tr>
<td>2</td>
<td>Joe, Max, John</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Abigail, Mark</td>
</tr>
</tbody>
</table>

0
1
2
3
Example: Computing the Hash Table Index

Storing names:

- Compute an integer from the name
- Map the integer to an index in a table
Hash Function

Hash function maps the keys to integers
Hash Function: Types

Mapping:

Map (a part of) the key into an integer

– Example: a letter to its position in the alphabet
Hash Function: Types

Folding:

Parts of the key combined by operations, such as add, multiply, shift, XOR, etc.

– Example: summing the values of each character in a string
Hash Function: Types

Shifting + Folding:

Shift left the name to get rid of repeating low-order bits or Shift right the name to multiply by powers of 2

Example: if keys are always even, shift off the low order bit
## Hash Function: Combinations

**Map, Fold, and Shift combination**

<table>
<thead>
<tr>
<th>Key</th>
<th>Mapped chars</th>
<th>Folded</th>
<th>Shifted and Folded</th>
</tr>
</thead>
<tbody>
<tr>
<td>eat</td>
<td>$5 + 1 + 20$</td>
<td>26</td>
<td>$20 + 2 + 20 = 42$</td>
</tr>
<tr>
<td>ate</td>
<td>$1 + 20 + 5$</td>
<td>26</td>
<td>$4 + 40 + 5 = 49$</td>
</tr>
<tr>
<td>tea</td>
<td>$20 + 5 + 1$</td>
<td>26</td>
<td>$80 + 10 + 1 = 91$</td>
</tr>
</tbody>
</table>
Hash Function: Types

Casts:

Converting a numeric type into an integer

– Example: casting a character to an integer to get its ASCII value
Hash Functions: Examples

– Key = Character:
  char value cast to an int \(\rightarrow\) its ASCII value

– Key = Date:
  value associated with the current time

– Key = Double:
  value generated by its bitwise representation
Hash Functions: Examples

– Key = Integer:
  the int value itself

– Key = String:
  a folded sum of the character values

– Key = URL:
  the hash code of the host name
Step 2: Mapping to a Valid Index

• Use modulus operator (%) with table size:
  – Example:

\[
idx = \text{hash}(\text{val}) \mod \text{size};
\]

• Must be sure that the final result is positive
  – Use only positive arithmetic or take absolute value
Step 2: Mapping to a Valid Index

To get a good distribution of indices, prime numbers make the best table sizes.

– Example: if you have 1000 elements, a table size of 997 or 1009 is preferable
Hash Tables: Ideal Case

1. **Perfect hash function**: each data element hashes to a unique hash index

2. **Table size equal to (or slightly larger than) number of elements**
Perfect Hashing: Example

- Six friends have a club: Alfred, Alessia, Amina, Amy, Andy, and Anne
- Store member names in a six element array
- Convert 3rd letter of each name to an index:

<table>
<thead>
<tr>
<th>Name</th>
<th>Index Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alfred</td>
<td>f = 5 % 6 = 5</td>
</tr>
<tr>
<td>Alessia</td>
<td>e = 4 % 6 = 4</td>
</tr>
<tr>
<td>Amina</td>
<td>i = 8 % 6 = 2</td>
</tr>
<tr>
<td>Amy</td>
<td>y = 24 % 6 = 0</td>
</tr>
<tr>
<td>Andy</td>
<td>d = 3 % 6 = 3</td>
</tr>
<tr>
<td>Anne</td>
<td>n = 13 % 6 = 1</td>
</tr>
</tbody>
</table>
Hash Tables: Collisions

• Unless the data is known in advance, the ideal case is usually not possible

• A collision is when two or more different keys result in the same hash table index

• How do we deal with collisions?
Indexing: Faster Than Searching

• Can convert a name (e.g., Alessia) into a number (e.g., 4) in constant time

• Faster than searching

• Allows for $O(1)$ time operations
Indexing: Faster Than Searching

Becomes complicated for new elements:

– Alan wants to join the club:
  ‘a’ = 0 $\rightarrow$ same as Amy

– Also:

  Al wants to join $\rightarrow$ no third letter!
Hash Tables: Resolving Collisions

There are two general approaches to resolving collisions:

1. **Open address hashing**: if a spot is full, probe for next empty spot

2. **Chaining (or buckets)**: keep a collection at each table entry
Open Address Hashing
Open Address Hashing

• All values are stored in an array

• Hash value is used to find initial index to try

• If that position is filled, next position is examined, then next, and so on until an empty position is filled
Open Address Hashing

• The process of looking for an empty position is termed *probing*,

• Specifically, we consider linear probing

• There are other probing algorithms, but we won’t consider them
Open Address Hashing: Example

Eight element table using the third-letter hash function:

Already added: Amina, Andy, Alessia, Alfred, and Aspen

<table>
<thead>
<tr>
<th>Amina</th>
<th>Andy</th>
<th>Alessia</th>
<th>Alfred</th>
<th>Aspen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>aiqy</td>
<td>bjrz</td>
<td>cks</td>
<td>dlt</td>
<td>emu</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fnv</td>
<td>gpw</td>
<td>hpq</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Open Address Hashing: Adding

Now we need to add: **Aimee**

The hashed index position (4) is filled by Alessia: so we probe to find next free location
Open Address Hashing: Adding

Suppose Anne wants to join:

Add: Anne  Hashes to

The hashed index position (5) is filled by Alfred:

Probe to find next free location

What happens when we reach the end of the array?
Open Address Hashing: Adding

Suppose Anne wants to join:

The hashed index position (5) is filled by Alfred:

- Probe to find next free location
- When we get to end of array, wrap around to the beginning
- Eventually, find position at index 1 open
Open Address Hashing: Adding

Finally, Alan wants to join:

The hashed index position (0) is filled by Amina:
– Probing finds last free position (index 2)
– Collection is now completely filled
Open Address Hashing: Contains

• Hash to find initial index, probe forward examining each location until value is found, or empty location is found.

• Example, search for: Amina, Aimee, Anne...

<table>
<thead>
<tr>
<th>Amina</th>
<th>Anne</th>
<th>Alan</th>
<th>Andy</th>
<th>Alessia</th>
<th>Alfred</th>
<th>Aimee</th>
<th>Aspen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>aiqy</td>
<td>bjrz</td>
<td>cks</td>
<td>dlt</td>
<td>emu</td>
<td>fnv</td>
<td>gpw</td>
<td>hpq</td>
</tr>
</tbody>
</table>

• Notice that search time is not uniform
Open Address Hashing: Remove

- Remove is tricky: Can’t leave this place empty
- What happens if we delete Anne, then search for Alan?

Remove: Anne

<table>
<thead>
<tr>
<th>Amina</th>
<th>Anne</th>
<th>Alan</th>
<th>Andy</th>
<th>Alessia</th>
<th>Alfred</th>
<th>Aimee</th>
<th>Aspen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-aiqy</td>
<td>1-bjrz</td>
<td>2-cks</td>
<td>3-dlt</td>
<td>4-emu</td>
<td>5-fnv</td>
<td>6-gpw</td>
<td>7-hpq</td>
</tr>
</tbody>
</table>

Find: Alan

Hashes to

Probing finds null entry ➔ Alan not found
Open Address Hashing: Handling Remove

• Replace removed item with “tombstone”
  – Special value that marks deleted entry
  – Can be replaced when adding new entry
  – But doesn’t halt search during contains (remove)

Find: Alan

Hashes to

Probing skips tombstone → Alan found

Amina TS Alan Andy Alessia Alfred Aimee Aspen

0-aiqy 1-bjrz 2-cks 3-dlt 4-emu 5-fnv 6-gpw 7-hpq
Hash Table Size: Load Factor

Load factor: \( \lambda = \frac{n}{m} \)

- Load factor is the average number of elements at each table entry
- For open address hashing, load factor is between 0 and 1 (often somewhere between 0.5 and 0.75)
- For chaining, load factor can be greater than 1
- Want the load factor to remain small
Large Load Factor: What to do?

• Common solution: When load factor becomes too large (say, bigger than 0.75) → Reorganize

• Create new table with twice the number of positions

• Copy each element, rehashing using the new table size, placing elements in new table

• Delete the old table
Hash Tables: Algorithmic Complexity

• Assumptions:
  – Time to compute hash function is constant
  – Worst case analysis \(\rightarrow\) All values hash to same position
  – Best case analysis \(\rightarrow\) Hash function uniformly distributes the values
Hash Tables: Algorithmic Complexity

• Find element operation:
  – Worst case for open addressing → O(n)
  – Best case for open addressing → O(1)
Hash Tables: Average Case

• What about average case?
• Turns out, it’s
  \[ \frac{1}{1 - \lambda} \]
• So keeping load factor small is very important

<table>
<thead>
<tr>
<th>λ</th>
<th>( \frac{1}{1 - \lambda} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.3</td>
</tr>
<tr>
<td>0.5</td>
<td>2.0</td>
</tr>
<tr>
<td>0.6</td>
<td>2.5</td>
</tr>
<tr>
<td>0.75</td>
<td>4.0</td>
</tr>
<tr>
<td>0.85</td>
<td>6.6</td>
</tr>
<tr>
<td>0.95</td>
<td>19.0</td>
</tr>
</tbody>
</table>
Difficulties with Hash Tables

• Need to find good hash function → uniformly distributes keys to all indices

• Open address hashing:
  – Need to tell if a position is empty or not
  – One solution → store only pointers

• Open address hashing: problem with removal