CPSC 311 Lecture Notes

Data Structures

(Chapters 10-12)

Acknowledgement: Parts of these course notes are based on notes from courses given by Jennifer Welch at Texas A&M University.

Data Structures for (Dynamic) Sets

Algorithms operate on data, which can be thought of as forming a **set** S. Unlike mathematical sets, the (data) sets manipulated by algorithms are **dynamic** – they can grow, shrink, or otherwise change over time.

Data Structures are structured ways to represent finite dynamic sets. Different data structures support different kinds of data manipulations, e.g.,

- **dictionary:** insert, delete, membership test
- **priority queue:** insert, extract-min

Operations on Dynamic Sets:

- INSERT(S, x) adds element pointed to by x to S
- DELETE(S, x) removes element pointed to by x from S.
- SEARCH(S, k) returns pointer to element x with $key[x] = k$ (or nil)
- MINIMUM(S) returns element with the smallest key
- MAXIMUM(S) returns element with the largest key
- SUCCESSOR(S, x) returns element with the next key larger than $key[x]$
- PREDECESSOR(S, x) returns element with the next key smaller than $key[x]$

Running Time: usually measure time of an operation in terms of the number of elements (currently) in the set.

Elementary Data Structures

These elementary data structures should be review from CPSC 211 (Chapt 10):

- **arrays** and **linked lists** (singly linked, doubly linked)
- **stacks** (e.g., arrays, lists)
- **queues** (e.g., arrays, lists)
- **rooted trees** (e.g., arbitrary trees using pointers, complete d-ary trees using arrays)

Binary Search Trees

Binary Search Trees should also be review from CPSC 211 (Chapt 12). Recall, every tree node (internal or leaf) contains a key.

Binary Search Tree Property: For every node x in tree

- $key[y] \leq key[x]$ for every y in $left(x)$ (left subtree)
- $key[y] \geq key[x]$ for every y in $right(x)$ (right subtree)

(Note there is no condition on the height h of tree)

Operations on BSTs (supports all dynamic data set ops)

- INSERT(S, x), DELETE(S, x)
- SEARCH(S, k), MINIMUM(S), MAXIMUM(S)
- SUCCESSOR(S, x), PREDECESSOR(S, x)

Running Time: All operations take $O(h)$ time

Traversing (visiting) BSTs

- **in-order:** visit $left(x)$, visit x, visit $right(x)$
- **pre-order:** visit x, visit $left(x)$, visit $right(x)$
- **post-order:** visit $left(x)$, visit $right(x)$, visit x

Binary Search Tree Operations

TREE-SEARCH(x, k)
  if $(x = NIL)$ or $(k = key[x])$
    return x
  if $(k < key[x])$
    then return TREE-SEARCH$(left[x], k)$
    else return TREE-SEARCH$(right[x], k)$

TREE-SUCCESSOR(x, k)
  if $right[x] \neq NIL$
    then return TREE-MINIMUM$(right[x])$
  $temp := parent[x]$
  while $(temp \neq NIL$ and $x = right[temp])$
    $x := temp$
    $temp := parent[temp]$
  return $temp$

Note:

- if x has a rightchild, then $successor(x)$ is the smallest node in the subtree rooted $right[x]$
- if x has no rightchild, then $successor(x)$ is the lowest ancestor of x whose leftchild is also an ancestor of x (or x itself)
- predecessor is symmetrical

Binary Search Tree Operations

INSERT(x, k)
  SEARCH(x, k) `/**stops at NIL, return 'parent'**/`
  insert x as leaf (child or returned parent)

DELETE(x, k)
  find x by SEARCH(x, k)
  if (x is a leaf)
    then delete x
  if (x has only one child)
    then 'splice' x out
  if (x has two children)
    1. find $successor(x)$ (it has at most one child)
    2. splice $successor(x)$ out of the tree
    3. replace x with $successor(x)$

Hash Tables

Dictionary operations

- SEARCH(k)
- INSERT(x)
- DELETE(x)

Definition:

U = Universe from which keys are drawn (often N, natural numbers)
$K \subseteq U$ set of keys seen

Goals:

- fast implementation of all operations — $O(1)$ time
- space efficient data structure — $O(n)$ space if n elements in dictionary

Approach 1: Linked Lists

Linked List Implementation

- INSERT(x): add x at head of list
- SEARCH(k): start at head and scan list
- DELETE(x): start at head, scan list, and then delete

Running Times: (assume n elements in list)

- INSERT(x): $O(1)$ time
- SEARCH(k):
  – worst-case, element at end of list: n operations
  – average-case, element at middle of list: $n/2$ operations
  – best-case, element at head of list: 1 operation
- DELETE(x): same as searching...

This is pretty bad.... we'd like $O(1)$ time on average for all operations....

Space Usage: (assume n elements in list)

- $O(n)$ space – very space efficient...

This is great!

Approach 2: Direct-Address Table

Direct-Address Table
Assume $U = \{0, 1, 2, \ldots, m\}$.
The data structure is an array $T[0, m]$.

- INSERT(x): $T[key[x]] := x$
- SEARCH(k): return $T[k]$
- DELETE(x): $T[key[x]] := NIL$

Running Times: (assume n elements in list)

- INSERT(x): $O(1)$ time
- SEARCH(x): $O(1)$ time
- DELETE(x): $O(1)$ time

This is great!

Space Usage: (assume n elements in list)

- $O(m)$ space always!
- good if $n = \Theta(m)$
- bad if $n << m$

Approach 3: Hashing

Hashing

- **hash table** (an array) $H[0, m]$ where $m << |U|$
  – amount of storage closer to what is really needed
- **hash function** h maps keys to indices in H
  – $h : U \rightarrow \{0, 1, \ldots, m\}$

Problem: there will be some **collisions**, that is, h will map some keys to the same position in H (e.g., $h(k_1) = h(k_2)$ for $k_1 \neq k_2$)

Different Methods of Resolving Collisions

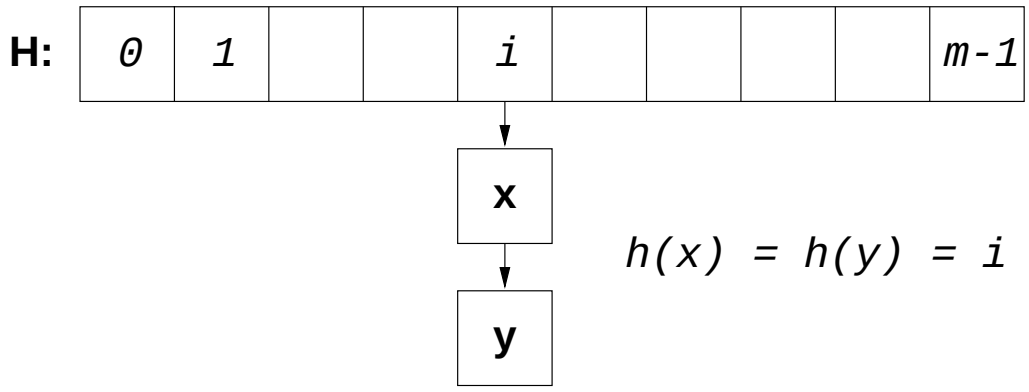
1. **chaining**: put all elements that hash to the same location in a linked list
2. **open addressing**: successively examine of **probe** H until find an empty slot (or the element you are looking for). There are various types of **probe sequences**:
   - *linear probing* – probe successive slots
   - *quadratic probing* – probes offset by quadratically increasing values
   - *double hashing* – a 2nd hash function determines probe sequence

Later we'll consider how to pick good hash functions h. Assume for now our hash function h satisfies:

Simple Uniform Hashing Assumption: Any key is equally likely to hash to any location (index,slot) in hash table H.

Collision Resolution by Chaining

Chaining: put all keys that hash to the same location in a linked list



- INSERT(x): compute $h(x)$ and insert x at head of linked list in $H[h(x)]$
  – $O(1)$ time always
- SEARCH(x): compute $h(x)$ and search linked list in $H[h(x)]$
  – $O(n)$ time in worst-case
- DELETE(x): compute $h(x)$ and search linked list $H[h(x)]$ for x
  – $O(n)$ time in worst-case
  – (can be $O(1)$ time if know where x is and list is doubly linked)

Note: even though **worst-case** behavior is $\Theta(n)$, the idea of hashing is to achieve good **average-case** behavior. Usually, we can get average-case time of operations down to $\Theta(1)$.

Average-Case Analysis for SEARCH(x) with Chaining

- let $H = [0, m-1]$
- let n be the number of elements currently in H
- $\alpha = \frac{n}{m}$ (the **load factor** of H)

Case 1: average SEARCH(x) time when x **is not** in table

- $= \Theta(1)$ to compute $h(x)$+ average time to examine list $H[h(x)]$ for x
- $= \Theta(1) + \Theta(\text{average length of list})$
- $= \Theta(1) + \Theta(\frac{n}{m})$
- $= \underline{\Theta(1+\alpha)}$

Average-Case Analysis for SEARCH(x) with Chaining

Case 2: average SEARCH(x) time when x **is** in table

- $= \Theta(1)$ to compute $h(x)$
  + expected # elements examined in $H[h(x)]$'s list until find x

Note: this is easier to analyze if we assume elements are added at end of lists (can prove average successful search time is the same regardless).

- $= \Theta(1)$ to compute $h(x)$
  + expected length of $H[h(x)]$ when x was inserted $+1$

Fact: if x is the ith element added to H, then the expected length of $H[h(x)]$'s list before adding x is $\frac{i-1}{m}$

$$
\begin{aligned}
&= \Theta(1) + \sum_{i=1}^{n} \Pr(x \text{ is } i\text{th elt added}) \cdot (\text{length of } H[h(x)] + 1) \\
&= \Theta(1) + \sum_{i=1}^{n} \frac{1}{n} \cdot \left(\frac{i-1}{m} + 1 \right) \\
&= \Theta(1) + \sum_{i=1}^{n} \frac{i-1}{nm} + \sum_{i=1}^{n} \frac{1}{n} \\
&= \Theta(1) + \frac{1}{nm} \sum_{i=1}^{n} (i-1) + \frac{1}{n} \sum_{i=1}^{n} 1 \\
&= \Theta(1) + \frac{1}{nm} \cdot \frac{(n-1)n}{2} + \frac{1}{n} \cdot n \\
&= \Theta(1) + \frac{n}{2m} - \frac{1}{2m} + 1 \\
&= \Theta(1) + \frac{1}{2} \cdot \alpha - \frac{1}{2m} + 1 = \Theta(1 + \alpha)
\end{aligned}
$$

So... average SEARCH(x) time is $\Theta(1 + \alpha)$
– $\Theta(1)$ if $\alpha = \Theta(1)$ (choose $m \geq n$ and $m = \Theta(n)$)

Choosing Hash Functions

Ideally, hash function satisfies:

Simple Uniform Hashing Assumption: Any key is equally likely to hash to any location (index,slot) in hash table H.

unfortunately, we cannot usually achieve this... so we use **heuristics**

For indexing in H is is convenient for keys to be natural numbers $(0, 1, 2, \ldots)$ – this is not usually a problem

- character strings $\Longrightarrow$ interpret characters as numbers
- real numbers $\Longrightarrow$ floor, ceiling (scale)

Division Method for creating hash functions

- $h(k) = k \bmod m$ (reminder when divide k by m)
- important to pick 'good' value for m, e.g., a **prime number** close to actual # of slots you want

Multiplicatoin Method for creating hash functions

- $h(k) = \lfloor m(kA \bmod 1) \rfloor$
- A is some constant $0 < A < 1$
- $kA \bmod 1$ is 'fractional part' of kA (i.e., $kA - \lfloor kA \rfloor$)
- the value selected for m is not as critical as for the division method – can get efficiency by choosing $m = 2^p$ for some constant p (p. 229)

Collision Resolution by Open Addressing

Main Idea:

- don't use linked list off hash table, put all elements in H
- successively examine or **probe** H until find x (or empty slot for it)
  – sequence in which slots are probed depends on key

hash function: includes probe number (try) as argument

- **probe sequence:** $h(k,0)$, $h(k,1)$, ..., $h(k,m-1)$
- examine every slot in worst-case
- stop when find element with key k (or empty slot)

What might happen if naively delete elements from H?

- might break 'sequence' and think we looked at all elts when we didn't
- book describes how to deal with deletions to avoide this problem (we won't cover in class)

Collision Resolution by Open Addressing

Ideally, we have

Uniform Hashing Assumption: each key is equally likely to have any of the $m!$ permutations (of indices in H) as its probe sequence

Note: this is different from **simple** uniform hashing:

- *simple uniform*: each key hashes to each of m slots with prob $\frac{1}{m}$
- *uniform*: each key hashes to each of $m!$ probe sequences with prob $\frac{1}{m!}$

Again, this is hard to achieve, so we usually settle for heurestics...

Linear Probing (Open Addressing)

Linear Probing: the ith probe $h(k,i)$ is

$$h(k,i) = (h'(k) + i) \bmod m$$

- $h'(k)$ is ordinary hash function, tells where to start the search
- search sequentially through table (with wrap around) from starting point

How many distinct probe sequences are there? **m**

- each starting point gives a probe sequence
- there are m starting points
- $\Longrightarrow$ **not uniform** (need $m!$ probe sequences)

Pluses and Minuses

- *plus:* easy to implement
- *minus:* leads to **clustering** (long run of occupied slots in H)
  – yields bad performance if hit cluster

Quadratic Probing (Open Addressing)

Quadratic Probing: the ith probe $h(k,i)$ is

$$h(k,i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$$

- c_1 and c_2 are constants
- $h'(k)$ is ordinary hash function, tells where to start the search
- later probes are offset by amount quadratic in i (the probe number)

How many distinct probe sequences are there? **m**

- each starting point gives a probe sequence
- there are m starting points
- $\Longrightarrow$ **not uniform** (need $m!$ probe sequences)

Pluses and Minuses

- *plus:* almost as easy to implement as linear probing
- *minus:* still leads to **clustering** but not as badly as linear probing (secondary clustering)

Double Hashing (Open Addressing)

Double Hashing: the ith probe $h(k, i)$ is

$$h(k, i) = (h_1(k) + h_2(k) \cdot i) \bmod m$$

- $h_1(k)$ is ordinary hash function, tells where to start the search
- $h_2(k)$ is ordinary hash function which gives offset for subsequent probes

Important: to make sure probe sequence hits all slots in H we must have $h_2(k)$ be relatively prime to m

Example 1:

- m is prime
- $h_1(k) = k \bmod m$
- $h_2(k) = 1 + (k \bmod (m-1))$

Example 2:

- m is power of 2
- $h_1(k) = k \bmod m$
- $h_2(k)$ is always odd

How many distinct probe sequences are there? m^2

- there are m starting points
- starting point and offset can vary independently
- better, but still not uniform...

Analyzing Open Addressing

We now analyze the expected number of probes for open addressing

- assume uniform hashing (all $m!$ probe sequences equally likely)
- $\alpha = \frac{n}{m}$ (load factor) so we need $\alpha \leq 1$ (table cannot be overfull)

Theorem: *If $\alpha < 1$, then the expected number of probes in an unsuccessful search is $\leq \frac{1}{1-\alpha}$*

Proof: In an unsuccessful search when $\alpha < 1$ (table not full), some number of probes access occupied slots and the last probe accesses an empty slot.

$$\begin{aligned}
E(\#\text{probes}) &= 1 + E(\#\text{probes that access occupied slots}) \\
&= 1 + \sum_{i=0}^{\infty} i \cdot \Pr[\text{exactly } i \text{ probes access occupied slots}] \\
&= 1 + \sum_{i=0}^{\infty} \Pr[\text{exactly } i \text{ probes access occupied slots}] \texttt{ /*identity*/} \\
&\leq 1 + \sum_{i=0}^{\infty} \alpha^i \texttt{/*proof below*/} \\
&= \frac{1}{1-\alpha}
\end{aligned}$$

□

Analyzing Open Addressing

Lemma: *Pr[at least i probes access occupied slots]* $\leq \alpha^i$

Proof:

$$\begin{aligned}
\Pr[\text{at least } i \text{ probes}] &= \Pr[\text{slot 1 full}] \cdot \Pr[\text{slot 2 full}] \cdot \ldots \cdot \Pr[\text{slot i-1 full}] \\
&= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \ldots \cdot \frac{n-(i-1)}{m-(i-1)} \\
&\leq \left(\frac{n}{m}\right)^i \\
&= \alpha^i
\end{aligned}$$

□

Thus, for example, we have:

- if hash table is half full ($\alpha = .5$), then the average number of probes in unsuccessful search is $\frac{1}{1-.5} = 2$.
- if hash table is 90% full ($\alpha = .9$), then the average number of probes in unsuccessful search is $\frac{1}{1-.9} = 10$.

Analyzing Open Addressing

Theorem: *If $\alpha < 1$, then the expected number of probes in an successful search is $\leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$*

Proof: Let k be the key being sought. Suppose k was the $(i+1)$st key inserted. The average # of probes needed to insert k was (by previous theorem):

$$\frac{1}{1 - \frac{i}{m}} = \frac{m}{m-i}$$

$$\begin{aligned}
E(\#\text{probes}) &= \sum_{i=0}^{n-1} \Pr[\text{k } (i+1)\text{st inserted}] \cdot (\#\text{probes used if k } (i+1)\text{st}) \\
&= \frac{1}{n} \cdot \sum_{i=0}^{n-1} \frac{m}{m-i} \\
&= \frac{m}{n} \cdot \sum_{i=0}^{n-1} \frac{1}{m-i} \\
&= \frac{1}{\alpha} \cdot \sum_{j=m-n+1}^{m} \frac{1}{j} \\
&\leq \frac{1}{\alpha} \cdot \int_{j=m-n}^{m} \frac{1}{x} dx \\
&= \frac{1}{\alpha} \cdot \ln \frac{m}{m-n} \\
&= \frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha}
\end{aligned}$$

□

Thus, for example, we have:

- if hash table is half full ($\alpha = .5$), then average number of probes in successful search is $.5 \ln 2 < 1$
- if hash table is 90% full ($\alpha = .9$), then average number of probes in successful search is $.9 \ln 10 \approx 2$

Exercise

1. Demonstrate (by picture) the insertion of the keys

$$5, 28, 19, 15, 20, 33, 12, 17, 10$$

into a hash table with collisions resolved by chaining. Let the table have $m = 9$ slots, and let the hash function be $h(k) = k \bmod m$.

2. Consider inserting the keys

$$10, 22, 31, 4, 15, 28, 17, 88, 59$$

into a hash table having $m = 11$ slots using open addressing with the primary hash function $h(k) = k \bmod m$. Illustrate (by picture) the result of inserting these keys using:

(a) linear probing

(b) quadratic probing with $c_1 = 1$ and $c_2 = 3$

(c) double hashing with $h_2(k) = 1 + (k \bmod (m - 1))$