The plan for this week

I’m going to review (since you should have seen it in CS211) some basic data structures:

• stacks
• queues
• linked lists
• trees

Then I’ll go into more details on hashing.

• You probably saw that in CS211 too, but I’ll cover it in more depth.

Stack Operations

Stack-Empty(S)
1 if top(S) = 0
2 then return True
3 else return False

Push(S,x)
1 top(S) ← top(S) + 1
2 S[top[S]] ← x

Pop(S)
1 if top(S) = 0 then return error “underflow”
2 top(S) ← top(S) − 1
3 return S[top(S) + 1]

• All these operations run in time O(1)

Stacks

Stacks support
• Insert = Push
• Delete(Maximum) = Pop
• test for emptiness: Stack-Empty

Stacks are implemented as arrays

• new elements are inserted at the end
• top[S] is the length of the array
• elements are retrieved from the end
  ○ LIFO: last in, first out

Queues

Queues support
• Insert = Enqueue
• Delete(Minimum) = Dequeue

Queues are implemented as arrays

• Have two indices: head and tail
• new elements are inserted at the tail
• elements are retrieved from the head
  ○ FIFO: first in, first out
Queue Operations

ENQUEUE(Q, x)
1 Q[tail(Q)] ← x
2 if tail(Q) = length(Q)
3 then tail(Q) ← 1  [wraparound]
4 else tail(Q) ← tail(Q) + 1

DEQUEUE(Q)
1 x ← Q[head(Q)]
2 if head(Q) = length(Q)
3 then head(Q) ← 1  [wraparound]
4 else head(Q) ← head(Q) + 1
5 return x

(We’re ignoring error conditions here.)

• ENQUEUE, DEQUEUE also run in O(1) time.

Linked Lists

There are many operations on dynamic sets that can’t be performed on Stacks and Queues (without implementing extra operations)

• E.g., searching, inserting

Linked lists are simple data structures that let us implement them all (not necessarily efficiently)

• doubly linked list: each entry contains a key, two pointers (next and prev), and perhaps other data
  ○ if next(x) = NIL then x has no successor
  ○ if prev(x) = NIL then x has no predecessor

• singly linked list: no prev pointer
  • head[L]/tail[L] is the first/last element of L;
    ○ can access L only by the head and tail
    ○ prev(head[L]) = next(tail[L]) = NIL

• circular list: next(tail[L]) = head[L];
  prev(head[L]) = tail[L]

Implementing Linked Lists

How do we implement linked lists in languages without pointers?

• Techniques useful even without pointers

Assuming no additional data, could use three arrays:

• key, next, prev

If keys have different sizes (or there is additional data), may be more efficient to use a single array:

• An entry is a contiguous part of the array A[j..k]

  • key is located at A[j], next pointer is in A[j + 1],
  • prev is in A[j + 2], rest of the data is in A[j + 3..k].

Allocation and Free Lists

Suppose we use an array (or several arrays) of length n to represent a linked list.

• Where in the array do we put a new element?

  • Can’t just use an initial segment of the array, because elements are getting deleted as well as inserted.

  If each record (element) takes a fixed amount of space, can use a free list to keep track of free slots in the array.

  • the free list is best implemented as a stack
    ○ POP a slot when you need to insert an element
    ○ PUSH a slot after its element has been deleted
Searching and Inserting in Linked Lists

To search a list for key $k$, start at the head and work towards the tail:

**LIST-SEARCH**($L$, $k$)
1 $x \leftarrow \text{head}[L]$
2 while $x \neq \text{NIL}$ and $\text{key}[x] \neq k$
3 do $x \leftarrow \text{next}[x]$
4 return $x$

If $k$ is not in the list, then we return NIL.

- Takes time $O(n)$ if $k$ is not in the list

Insert a new element at the head:

**LIST-INSERT**($L$, $x$)
1 $\text{next}[x] \leftarrow \text{head}[L]$
2 if $\text{head}[L] \neq \text{NIL}$ [list is not empty]
3 then $\text{prev}[\text{head}[L]] \leftarrow x$
4 $\text{head}[L] \leftarrow x$
5 $\text{prev}[x] \leftarrow \text{NIL}$

Deletion in Linked Lists

To delete $x$, edit it out of the list:

**LIST-DELETE**($L$, $x$)
1 if $\text{prev}[x] \neq \text{NIL}$
2 then $\text{next}[	ext{prev}[x]] \leftarrow \text{next}[x]$
3 else $\text{head}[L] \leftarrow \text{next}[x]$
4 if $\text{next}[x] \neq \text{NIL}$
5 then $\text{prev}[\text{next}[x]] \leftarrow \text{prev}[x]$

Deletion takes $O(1)$ for doubly-linked lists

- It’s important here that $x$ is a pointer, not a key
- If it’s a key, deletion take $O(n)$

Deletion takes $O(n)$ for singly-linked lists

- Problem: need to find the predecessor of $x$ so that $\text{next}[\text{predecessor}]$ can be set to $\text{next}[x]$.

Representing Rooted Trees

Suppose we have a (rooted) binary tree. Then can use something like a linked list:

- $\text{head}$ points to the root
- $\text{prev}[x]$ points to the (unique) parent of $x$
- instead of $\text{next}$, have left-child and right-child
  - $x$ has two successors, not one

Similar ideas work for $k$-ary trees, if $k$ is bounded.

What happens if we have no bound on the branching factor of the tree?

- Hard to allocate space upfront if we represent each child explicitly
- Even if we have an upper bound of $k$, but most nodes have fewer than $k$ children, there will be lots of wasted space.

Left-child Right-sibling representation

**Left-child right-sibling representation**

- This uses only $O(n)$ space for an $n$-node tree.
Direct-Address Tables

Suppose we want to implement a dictionary

- **INSERT, DELETE, SEARCH**

Assume keys are drawn from \{0, 1, \ldots, m - 1\}
- \(m\) is “not too large”
- all keys distinct

Can just use an array \(T[0..m - 1]\)
- \(T[k]\) points to element with key \(k\)
- \(T[k] = \text{NIL}\) if there is no element with key \(k\)
- insertion, deletion, and search are all trivial
  - \(O(1)\) worst-case time

**Problem:** what happens if \(m\) is large?
- storing a table of size \(m\) may be impractical (or impossible)

Hash Tables

The idea of using \(\text{key}[x]\) to determine where \(x\) is stored is good.
- Keys are drawn from universe \(U\)
- **Hash function** \(h : U \rightarrow \{1, \ldots, m\}\)
  - \(k\) hashes to \(h(k)\)
- Array has length \(m\) instead of \(|U|\)
  - Problem: What happens of \(h(k) = h(k')\)? A **collision**!

- A good hash function minimizes the chances of collisions
  - Can’t avoid them altogether if \(|U| > m\)
- A good implementation of hashing minimizes the impact of collisions

Collision Resolution by Chaining

In chaining, put all the elements that hash to the same slot in a linked list.
- Slot \(j\) has a pointer to the head of a linked list containing all the elements that hash to \(j\)
- If there aren’t any elements that hash to \(j\), slot \(j\) contains NIL.

Simple algorithms for dictionary operations:

**Chained-Hash-Insert**(*T*, *x*)
1. insert \(x\) at the head of list \(T[h(\text{key}[x])]\)

**Chained-Hash-Search**(*T*, *k*)

Basically just linked-list search (see **List-Search**(*L*, *k*))
1. \(y \leftarrow T[h(k)]\) \quad \(T[h(k)]\) is the head of the linked list
2. while \(y \neq \text{NIL}\) or \(\text{key}[y] \neq k\)
3. do \(y \leftarrow \text{next}[y]\)
4. return \(y\)

**Chained-Hash-Delete**(*T*, *x*)
1. delete \(x\) from the list \(T[h(\text{key}[x])]\)
Analysis of Hashing with Chaining

If a table $T$ has $m$ slots and $n$ keys are stored, the \textit{load factor} of $T$ is $\alpha = n/m$:

\begin{itemize}
  \item the average number of elements per slot
  \item the average number of elements in a list
\end{itemize}

The worst-case behavior of hashing is like that of linked lists:

\begin{itemize}
  \item happens if all keys are hashed to the same slot
\end{itemize}

Assume that each element is equally likely to hash into any slot.

\begin{itemize}
  \item \textit{simple uniform hashing}
\end{itemize}

\textbf{Theorem}: Using hashing with chaining, a search (successful or unsuccessful) takes time $O(\alpha + 1)$ on average, assuming simple uniform hashing.

\textbf{Proof}: Every key is equally likely to hash to any slot.

\begin{itemize}
  \item the average length of a list is $\alpha$
  \item in an unsuccessful search, we need to look at all of them
  \item in a successful search, on average, we look at half of them
\end{itemize}

If $n = O(m)$, then $\alpha = O(1)$ and searching is fast.

\begin{itemize}
  \item Hashing is great for dictionary operations
  \item Not so good for max and min
\end{itemize}

Choosing a Good Hash Function

We want a hash function for which each key is equally likely to hash to any slot \textit{no matter how keys are distributed}.

\begin{itemize}
  \item E.g.: if keys are identifiers in a program, closely related symbols are likely to occur (pt and pts)
\end{itemize}

Sometimes want keys that are “close” to yield hash values that are far apart.

The Division Method

\textbf{Assumption}: All keys are natural numbers.

\begin{itemize}
  \item Can convert names to numbers using a standard translation
\end{itemize}

\textbf{Division Method}: $h(k) = k \mod m$

\begin{itemize}
  \item if $m = 12$, then $h(100) = h(16) = 4$
\end{itemize}

Bad choices for $m$:

\begin{itemize}
  \item $m = 2^p$ means that $h(k)$ is the $p$ lower-order bits (if $k$ is written base 2)
    \begin{itemize}
      \item can be bad if not all patterns equally likely
    \end{itemize}
  \item $m = 10^p$ is bad if $k$ is written base 10
\end{itemize}

Good choice for $m$: a prime number

\begin{itemize}
  \item If you have an estimate $n$ for $|U|$, and a tolerable load factor $\alpha$, choose a prime $m \sim n/\alpha$
\end{itemize}
The Multiplication Method

The Multiplication Method:
\[ h(k) = \lfloor m(kA \mod 1) \rfloor \]

Explanation:
1. Choose a fixed constant \( A \) with \( 0 < A < 1 \), compute \( kA \)
2. \( kA \mod 1 \) is the fractional part of \( kA \)
3. multiply this by \( m \) and take the floor of the answer

Example: Suppose \( A = 7/10, m = 5 \)
- \( h(117) = \lfloor 5(819/10 \mod 1) \rfloor = \lfloor 5(9/10) \rfloor = 4 \)

Almost any choice of \( A \) and \( m \) will work but ...
- Choosing \( m \) a power of 2 \( (m = 2^p) \) makes for easy implementation
- Choose \( A \) so that, if rational, its denominator is > \( m \)
- Knuth suggests \( A \approx (\sqrt{5} - 1)/2 \)

Universal Hashing

If I know your hash function, then I can choose \( n \) keys that all hash to the same slot.

Better idea:
- Choose the hash function randomly, so that no malicious adversary can foil you
- That’s what universal hashing [Carter-Wegman] is all about

Formally, let \( \mathcal{H} \) be a set of hash functions.
- \( \mathcal{H} \) is universal if, for all \( x, y \), the number of hash functions \( h \) such that \( h(x) = h(y) \) is \( |\mathcal{H}|/m \)
- Therefore, if \( h \in \mathcal{H} \) is chosen randomly, the probability that \( h(x) = h(y) \) is \( 1/m \)
  - \( 1/m \) functions cause a collision, \((m - 1)/m \) don’t
- This is exactly the chance of a collision if \( h(x) \) and \( h(y) \) are chosen randomly from \( \{0, \ldots, m - 1\} \)

Universal hashing is good even if we don’t assume that the inputs are uniformly distributed.

Theorem: If \( h \in \mathcal{H} \) is chosen randomly and is used to hash \( n \) keys into a table of size \( m \), the expected \# of collisions involving \( x \) is \((n - 1)/m \).

Proof: Let \( C_{yz} \) be a random variable (on \( \mathcal{H} \)) such that
- \( C_{yz}(h) = 1 \) if \( h(y) = h(z) \), 0 otherwise

Since \( \mathcal{H} \) is universal, \( E(C_{yz}) = 1/m \)

Let \( C_x \) be the total \# of collisions involving \( x \):
\[ C_x = \sum_{y \neq z} C_{yz} \]
\[ E(C_x) = \sum_{y \neq z} E(C_{yz}) = (n - 1)/m \]

Are there universal classes of hash functions?
If so, how hard are they to implement?

Not hard, if we assume a known upper bound on key size:
- Let \( m \) be prime.
- Suppose \( k \) can be written as \((k_0, \ldots, k_r)\) for some \( r \), where \( 0 \leq k_i \leq r \)
- Hash function has form \( h_{(a_0, \ldots, a_r)}(k_0, \ldots, k_r) = \sum_{i=0}^r a_i k_i \)
  - There are \( m^{r+1} \) such functions

Theorem: This set of hash functions is universal.
Open Addressing

Idea of open addressing:

- all elements are stored in the hash table
- no pointers, no linked lists
- by not having pointers, can afford to have a larger hash table

So where do we put elements if there is a collision?

- Idea: have first choice, second choice, etc.
- Probe the hash table until we find a free slot

Formally, to hash from $U$ to $\{0, \ldots, m - 1\}$, consider hash functions of the form:

$$h : U \times \{0, \ldots, m - 1\} \to \{0, \ldots, m - 1\}$$

- $h(k, j)$ is $(j + 1)$th place to look for/insert key $k$
- Want $h(k, 0), \ldots, h(k, m - 1)$ to all be different
  - $(h(k, 0), \ldots, h(k, m - 1))$ is a permutation of $\{0, \ldots, m - 1\}$