Dynamic sets

- Sets are fundamental to computer science
- Algorithms may require several different types of operations to be performed on sets
- For example, many algorithms need only the ability to
  - insert elements into, delete elements from, and test membership in a set
- We call a dynamic set that supports these operations a *dictionary*
Operations on dynamic sets

- Operations on a dynamic set can be grouped into **queries** and **modifying operations**

**SEARCH**(*S*, *k*)
- Given a set *S* and a key value *k*, return a pointer to an element in *S* such that *x*.key = *k*, or **NIL** if no such element belongs to *S*

**INSERT**(*S*, *x*)
- Augment the set *S* with the element pointed to by *x*. We assume that any attributes in element *x* have already been initialized

**DELETE**(*S*, *x*)
- Given a pointer *x* to an element in the set *S*, remove *x* from *S*. Note that this operation takes a pointer to an element *x*, not a key value

**MINIMUM**(*S*)
- A query on a totally ordered set *S* that returns a pointer to the element of *S* with the smallest key

**MAXIMUM**(*S*)
- Return a pointer to the element of *S* with the largest key

**SUCCESSOR**(*S*, *x*)
- Given an element *x* whose key is from a totally ordered set *S*, return a pointer to the next larger element in *S*, or **NIL** if *x* is the maximum element

**PREDECESSOR**(*S*, *x*)
- Given an element *x* whose key is from a totally ordered set *S*, return a pointer to the next smaller element in *S*, or **NIL** if *x* is the minimum element

- We usually measure the time taken to execute a set operation in terms of the size of the set
- E.g., we later describe a data structure that can support any of the operations on a set of size *n* in time \(O(\log n)\)
10 Elementary Data Structures

10.1 Stacks and queues

- In stacks and queues the element removed from the set by DELETE is prespecified.
- In a stack, the element deleted is the one most recently inserted: the stack implements a last-in, first-out (LIFO) policy.
- In a queue the element deleted is the one that has been in the set for the longest time: the queue policy is first-in, first-out (FIFO).
- We consider array to implementation of stacks and queues.

Stacks

- INSERT operation on a stack is called PUSH.
- DELETE operation, which does not take an element argument, is called POP.
- These names are allusions to physical stacks.
We can implement a stack of at most \( n \) elements with an array \( S[1..n] \).

The array has an attribute \( S.top \) that indexes the most recently inserted element.

The stack consists of elements \( S[1..S.top] \), where \( S[1] \) is the element at the bottom of the stack and \( S[S.top] \) is the element at the top.

When \( S.top = 0 \), the stack is empty.

We can test to see whether the stack is empty by query operation \( \text{STACK-EMPTY} \).

Popping an empty stack leads to underflow.

---

**STACK-EMPTY(S)**

1. if \( S.top == 0 \)
2. return TRUE
3. else return FALSE

**POP(S)**

1. if \( \text{STACK-EMPTY}(S) \)
2. error “underflow”
3. else \( S.top = S.top - 1 \)
4. return \( S[S.top + 1] \)

**PUSH(S, x)**

1. \( S.top = S.top + 1 \)
2. \( S[S.top] = x \)

Each stack operation takes \( O(1) \) time.
Queues

- The **INSERT** operation on a queue is **ENQUEUE**
- We call the **DELETE** operation **DEQUEUE**
  - like the stack operation **POP**, **DEQUEUE** takes no element argument
- The queue has a **head** and a **tail**
- When an element is enqueued, it takes its place at the tail of the queue
- The element dequeued is always the one at the head of the queue
One way to implement a queue of at most $n - 1$ elements using an array $Q[1..n]$

- The queue has an attribute $Q\.head$ that indexes (points) to its head
- The attribute $Q\.tail$ indexes the next location at which a newly arriving element will be inserted into the queue
- We “wrap around”: location 1 immediately follows location $n$ in a circular order
- When $Q\.head = Q\.tail$, the queue is empty
- Initially, we have $Q\.head = Q\.tail = 1$

**ENQUEUE($Q,x$)**

1. $Q[Q\.tail] = x$
2. if $Q\.tail == Q\.length$
3. $Q\.tail = 1$
4. else
   $Q\.tail = Q\.tail + 1$

**DEQUEUE($Q$)**

1. $x = Q[Q\.head]$
2. if $Q\.head == Q\.length$
3. $Q\.head = 1$
4. else
   $Q\.head = Q\.head + 1$
5. return $x$

Both operations take $O(1)$ time
10.2 Linked lists

- The linear order in a linked list is determined by a pointer in each object.
- Each element of a doubly linked list $L$ is an object with an attribute `key` and two other pointer attributes: `next` and `prev`.
- $x.next$ points to successor in the linked list, and $x.prev$ points to predecessor.
- If $x.prev = \text{NIL}$, the element $x$ is the first element, or `head`, of the list.
- If $x.next = \text{NIL}$, the element $x$ is the last element, or `tail`, of the list.
- Attribute $L.head$ points to the first element of the list.
- If $L.head = \text{NIL}$, the list is empty.
Searching a linked list

\textsc{List-Search} \,(L, k) \\
1. \, x = L.\text{head} \\
2. \, \textbf{while} \, x \neq \text{NIL} \, \textbf{and} \, x.\text{key} \neq k \\
3. \, x = x.\text{next} \\
4. \, \textbf{return} \, x \\

- To search a list of \( n \) objects, \textsc{List-Search} takes \( \Theta(n) \) time in the worst case, since it may have to search the entire list

Inserting into a linked list

\textsc{List-Insert} \,(L, x) \\
1. \, x.\text{next} = L.\text{head} \\
2. \, \textbf{if} \, L.\text{head} \neq \text{NIL} \\
3. \, L.\text{head}.\text{prev} = x \\
4. \, L.\text{head} = x \\
5. \, x.\text{prev} = \text{NIL} \\

- The running time for \textsc{List-Insert} is \( O(1) \)
Deleting from a linked list

LIST-DELETE\((L, x)\)

1. if \(x.\text{prev} \neq \text{NIL}\)
2. \(x.\text{prev}.\text{next} = x.\text{next}\)
3. else \(L.\text{head} = x.\text{next}\)
4. if \(x.\text{next} \neq \text{NIL}\)
5. \(x.\text{next}.\text{prev} = x.\text{prev}\)

• This runs in \(O(1)\) time, if we wish to delete an element with a given key, \(\Theta(n)\) time is required

Sentinels

• LIST-DELETE would be simpler if we could ignore the boundary conditions at the head and tail of the list:

LIST-DELETE\(^{'}\)(\(L, x\))

1. \(x.\text{prev}.\text{next} = x.\text{next}\)
2. \(x.\text{next}.\text{prev} = x.\text{prev}\)
A sentinel is a dummy object that allows us to simplify boundary conditions.

We provide with list \( L \) an object \( L.\text{nil} \) that represents \( \text{NIL} \) but has all the attributes of the other objects in the list.

A reference to \( \text{NIL} \) in list code is replaced by a reference to the sentinel \( L.\text{nil} \).

This turns a doubly linked list into a circular, doubly linked list with a sentinel, the sentinel \( L.\text{nil} \) lies between the head and tail.

- Attribute \( L.\text{nil}.\text{next} \) points to the head of the list, and \( L.\text{nil}.\text{prev} \) points to the tail.

- Similarly, both the \text{next} attribute of the tail and the \text{prev} attribute of the head point to \( L.\text{nil} \).

- Since \( L.\text{nil}.\text{next} \) points to the head, we can eliminate the attribute \( L.\text{head} \) altogether.
• The code for LIST-SEARCH remains the same as before, but references to \texttt{NIL} and \texttt{L.head} change:

\textbf{LIST-SEARCH'}(L, k)

1. $x = L.\texttt{nil.next}$
2. while $x \neq L.\texttt{nil}$ and $x.key \neq k$
3. $x = x.\texttt{next}$
4. return $x$

• Sentinels rarely reduce the asymptotic time bounds, but can reduce constant factors
• The gain is usually a matter of clarity of code rather than speed
• In other situations, however, the use of sentinels helps to tighten the code in a loop, thus reducing the coefficient of, say, $n$ or $n^2$ in the running time
• Sentinels should be used judiciously:
  – When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory
11 Hash Tables

- Often one only needs the dictionary operations INSERT, SEARCH, and DELETE
- Hash table effectively implements dictionaries
- In the worst case, searching for an element in a hash table takes $\Theta(n)$ time
- In practice, hashing performs extremely well
- Under reasonable assumptions, the average time to search for an element is $O(1)$

11.2 Hash tables

- A set $K$ of keys is stored in a dictionary, it is usually much smaller than the universe $U$ of all possible keys
- A hash table requires storage $\Theta(|K|)$ while search for an element in it only takes $O(1)$ time
- The catch is that this bound is for the average-case time
An element with key $k$ is stored in slot $h(k)$; that is, we use a hash function $h$ to compute the slot from the key $k$.

- $h: U \rightarrow \{0,1,\ldots,m-1\}$ maps the universe $U$ of keys into the slots of hash table $T[0..m-1]$.
- The size $m$ of the hash table is typically $\ll |U|$.
- We say that an element with key $k$ hashes to slot $h(k)$ or that $h(k)$ is the hash value of $k$.
- If $k_1 \neq k_2$ hash to same slot we have a collision.
- Effective techniques resolve the conflict.
The ideal solution avoids collisions altogether

We might try to achieve this goal by choosing a suitable hash function $h$

One idea is to make $h$ appear to be random, thus minimizing the number of collisions

Of course, $h$ must be deterministic so that a key $k$ always produces the same output $h(k)$

Because $|U| > m$, there must be at least two keys that have the same hash value;
– avoiding collisions altogether is therefore impossible

Collision resolution by chaining

In chaining, we place all the elements that hash to the same slot into the same linked list

Slot $j$ contains a pointer to the head of the list of all stored elements that hash to $j$

If no such elements exist, slot $j$ contains NIL

The dictionary operations on a hash table $T$ are easy to implement when collisions are resolved by chaining
**CHAINED-HASH-INSERT** \((T, x)\)
1. insert \(x\) at the head of list \(T[h(x.key)]\)

**CHAINED-HASH-SEARCH** \((T, k)\)
1. search for element with key \(k\) in list \(T[h(k)]\)

**CHAINED-HASH-DELETE** \((T, x)\)
1. delete \(x\) from the list \(T[h(x.key)]\)

- Worst-case running time of insertion is \(O(1)\)
- It is fast in part because it assumes that the element \(x\) being inserted is not already present in the table
- For searching, the worst-case running time is proportional to the length of the list;
  - we analyze this operation more closely soon
- We can delete an element (given a pointer) in \(O(1)\) time if the lists are doubly linked
- In singly linked lists, to delete \(x\), we would first have to find \(x\) in the list \(T[h(x.key)]\)
Analysis of hashing with chaining

- A hash table $T$ of $m$ slots stores $n$ elements, define the load factor $\alpha = n/m$, i.e., the average number of elements in a chain
- Our analysis will be in terms of $\alpha$, which can be $< 1$, $= 1$, or $> 1$
- In the worst-case all $n$ keys hash to one slot
- The worst-case time for searching is thus $\Theta(n)$ plus the time to compute the hash function

Average-case performance of hashing depends on how well $h$ distributes the set of keys to be stored among the $m$ slots, on the average

- **Simple uniform hashing** (SUH):
  - Assume that an element is equally likely to hash into any of the $m$ slots, independently of where any other element has hashed to
  - For $j = 0, 1, \ldots, m - 1$, let us denote the length of the list $T[j]$ by $n_j$, so $n = n_0 + \cdots + n_{m-1}$
  - and the expected value of $n_j$ is
    $$\mathbb{E}[n_j] = \alpha = n/m$$
**Theorem 11.1**  
*In a hash table which resolves collisions by chaining, an unsuccessful search takes average-case time $\Theta(1 + \alpha)$, under SUH assumption.*

**Proof**  
Under SUH, $k$ not already in the table is equally likely to hash to any of the $m$ slots. The time to search unsuccessfully for $k$ is the expected time to go through list $T[h(k)]$, which has expected length $E[n_{h(k)}] = \alpha$. Thus, the expected number of elements examined is $\alpha$, and the total time required (including computing $h(k)$) is $\Theta(1 + \alpha)$. ■

---

**Theorem 11.2**  
*In a hash table which resolves collisions by chaining, a successful search takes average-case time $\Theta(1 + \alpha)$, under SUH.*

**Proof**  
We assume that the element being searched for is equally likely to be any of the $n$ elements stored in the table. The number of elements examined during the search for $x$ is one more than the number of elements that appear before $x$ in $x$’s list. Elements before $x$ in the list were all inserted after $x$ was inserted.
Let us take the average (over the \( n \) table elements \( x \)) of the expected number of elements added to \( x \)'s list after \( x \) was added to the list + 1.

Let \( x_i, i = 1,2,\ldots,n \), denote the \( i \)th element inserted into the table and let \( k_i = x_i.key \).

For keys \( k_i \) and \( k_j \), define the indicator random variable \( X_{ij} = I\{h(k_i) = h(k_j)\} \).

Under SUH, \( \Pr\{h(k_i) = h(k_j)\} = 1/m \), and by Lemma 5.1, \( E[X_{ij}] = 1/m \).

Thus, the expected number of elements examined in successful search is

\[
E \left[ \frac{1}{n} \sum_{i=1}^{n} \left( 1 + \sum_{j=i+1}^{n} X_{ij} \right) \right] \\
= \frac{1}{n} \sum_{i=1}^{n} \left( 1 + \sum_{j=i+1}^{n} E[X_{ij}] \right) \\
= \frac{1}{n} \sum_{i=1}^{n} \left( 1 + \sum_{j=i+1}^{n} \frac{1}{m} \right)
\]
Thus, the total time required for a successful search is $\Theta(2 + \alpha/2 + \alpha/2n) = \Theta(1 + \alpha)$.

- If the number of hash-table slots is at least proportional to the number of elements in the table, we have $n = O(m)$ and, consequently, $\alpha = n/m = O(m)/m = O(1)$.
- Thus, searching takes constant time on average.
- Insertion takes $O(1)$ worst-case time.
- Deletion takes $O(1)$ worst-case time when the lists are doubly linked.
- Hence, we can support all dictionary operations in $O(1)$ time on average.
11.3 Hash functions

- A good hash function satisfies (approximately) the assumption of SUH:
  - each key is equally likely to hash to any of the $m$ slots, independently of the other keys
- We typically have no way to check this condition, since we rarely know the probability distribution from which the keys are drawn
- The keys might not be drawn independently

- If we, e.g., know that the keys are random real numbers $k$ independently and uniformly distributed in the range $0 \leq k < 1$
- Then the hash function $h(k) = [km]$ satisfies the condition of SUH
- In practice, we can often employ heuristic techniques to create a hash function that performs well
- Qualitative information about the distribution of keys may be useful in this design process
In a compiler’s symbol table the keys are strings representing identifiers in a program.

Closely related symbols, such as `pt` and `pts`, often occur in the same program.

A good hash function minimizes the chance that such variants hash to the same slot.

A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data.

For example, the “division method” computes the hash value as the remainder when the key is divided by a specified prime number.

### Interpreting keys as natural numbers

- Hash functions assume that the universe of keys is natural numbers $\mathbb{N} = \{0, 1, 2, \ldots \}$.
- We can interpret a character string as an integer expressed in suitable radix notation.
- We interpret the identifier `pt` as the pair of decimal integers $(112, 116)$, since $p = 112$ and $t = 116$ in ASCII.
- Expressed as a radix-128 integer, `pt` becomes $112 \cdot 128 + 116 = 14,452$. 

11.3.1 The division method

- In the division method, we map a key $k$ into one of $m$ slots by taking the remainder of $k$ divided by $m$
- That is, the hash function is $h(k) = k \mod m$
- E.g., if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$
- Since it requires only a single division operation, hashing by division is quite fast

- When using the division method, we usually avoid certain values of $m$
- E.g., $m$ should not be a power of $2$, since if $m = 2^p$, then $h(k)$ is just the $p$ lowest-order bits of $k$
- We are better off designing the hash function to depend on all the bits of the key
- A prime not too close to an exact power of $2$ is often a good choice for $m$
Suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly \( n = 2,000 \) character strings, where a character has 8 bits.

We don’t mind examining an average of 3 elements in an unsuccessful search, and so we allocate a hash table of size \( m = 701 \).

We choose \( m = 701 \) because it is a prime near \( 2000/3 \) but not near any power of 2.

Treating each key \( k \) as an integer, our hash function would be \( h(k) = k \mod 701 \).

### 11.3.2 The multiplication method

The method operates in two steps:

- First, multiply \( k \) by a constant \( A, \ 0 < A < 1 \), and extract the fractional part of \( kA \).
- Then, multiply this value by \( m \) and take the floor of the result.

The hash function is

\[
h(k) = \lfloor m(kA \mod 1) \rfloor
\]

where \( kA \mod 1 \) means the fractional part of \( kA \), that is, \( kA - \lfloor kA \rfloor \).
• Value of $m$ is not critical, we choose $m = 2^p$

• The function is implemented on computers
  – Suppose that the word size of the machine is $w$ bits and that $k$ fits into a single word
  – Restrict $A$ to be a fraction of the form $s/2^w$, where $s$ is an integer in the range $0 < s < 2^w$
  – Multiply $k$ by the $w$-bit integer $s = A \cdot 2^w$
  – The result is a $2w$-bit value $r_1 2^w + r_0$, where $r_1$ is the high-order word of the product and $r_0$ is the low-order word of the product
  – The desired $p$-bit hash value consists of the $p$ most significant bits of $r_0$
• Knuth (1973) suggests that
  \[ A \approx (\sqrt{5} - 1)/2 = 0.6180339887 \ldots \]
is likely to work reasonably well
• Suppose that \( k = 123,456 \), \( p = 14 \), \( m = 2^{14} = 16,384 \), and \( w = 32 \)
• Let \( A \) to be the fraction of the form \( s/2^{32} \) closest to \( (\sqrt{5} - 1)/2 \), so that \( A = 2,654,435,769/2^{32} \)
• Then
  \[ k \cdot s = 327,706,022,297,664 = (76,300 \cdot 2^{32}) + 17,612,864, \text{ and } r_1 = 76,300 \text{ and } r_0 = 17,612,864 \]
• The 14 most significant bits of \( r_0 \) yield the value \( h(k) = 67 \)

11.4 Open addressing

• In open addressing, all elements occupy the hash table itself
• That is, each table entry contains either an element of the dynamic set or NIL
• Search for element systematically examines table slots until we find the element or have ascertained that it is not in the table
• No lists and no elements are stored outside the table, unlike in chaining
Thus, the hash table can “fill up” so that no further insertions can be made
– the load factor $\alpha$ can never exceed 1
We could store the linked lists for chaining inside the hash table, in the otherwise unused hash-table slots
Instead of following pointers, we compute the sequence of slots to be examined
The extra memory freed by provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval

To perform insertion, we successively examine, or **probe**, the hash table until we find an empty slot in which to put the key
Instead of being fixed in order $0, 1, \ldots, m - 1$ (which requires $\Theta(n)$ search time), the sequence of positions probed depends upon the key being inserted
To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input
Thus, the hash function becomes

\[ h: U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\} \]

With open addressing, we require that for every key \( k \), the probe sequence

\[ h(k,0), h(k,1), \ldots, h(k,m-1) \]

be a permutation of \( \{0,1,\ldots,m-1\} \), so that every position is eventually considered as a slot for a new key as the table fills up

Let us assume that the elements in the hash table \( T \) are keys with no satellite information; the key \( k \) is identical to the element containing key \( k \)

**HASH-INSERT** either returns the slot number of key \( k \) or flags an error because the table is full

**HASH-INSERT** \( (T, k) \)

1. \( i = 0 \)
2. repeat
3. \( j = h(k, i) \)
4. if \( T[j] == \text{NIL} \)
5. \( T[j] = k \)
6. return \( j \)
7. else \( i = i + 1 \)
8. until \( i == m \)
9. error “hash table overflow”
• Search for key $k$ probes the same sequence of slots that the insertion algorithm examined

**HASH-SEARCH**$(T, k)$

1. $i = 0$
2. repeat
3. $j = h(k, i)$
4. if $T[j] == k$
5. return $j$
6. $i = i + 1$
7. until $T[j] == \text{NIL}$ or $i == m$
8. return $\text{NIL}$

• When we delete a key from slot $i$, we cannot simply mark it as empty by storing $\text{NIL}$ in it
  – We might be unable to retrieve any key $k$ during whose insertion we had probed slot $i$
• Instead we mark the slot with value $\text{DELETED}$
• Modify **HASH-INSERT** to treat such a slot as empty so that we can insert a new key there
• **HASH-SEARCH** passes over $\text{DELETED}$ values
• When we use $\text{DELETED}$ value, search times no longer depend on the load factor $\alpha$
• Therefore chaining is more commonly selected as a collision resolution technique