• Worst-case running time for RANDOMIZED-SELECT is $\Theta(n^2)$, even to find the minimum
• The algorithm has a linear expected running time, though, and because it is randomized, no particular input elicits the worst-case behavior
• Let the running time of RANDOMIZED-SELECT on an input array $A[p..r]$ of $n$ elements be a random variable $T(n)$; we obtain an upper bound on $E[T(n)]$ as follows
• The procedure RANDOMIZED-PARTITION is equally likely to return any element as the pivot

Therefore, for each $k$ such that $1 \leq k \leq n$, the subarray $A[p..q]$ has $k$ elements (all less than or equal to the pivot) with probability $1/n$
• For $k = 1, 2, ..., n$, define indicator random variables $X_k$ where
  $$X_k = I\{\text{the subarray } A[p..q] \text{ has exactly } k \text{ elements}\}$$
• Assuming that the elements are distinct, we have $E[X_k] = 1/n$
• When we choose $A[q]$ as the pivot element, we do not know, a priori, 1) if we will terminate immediately with the correct answer, 2) recurse on the subarray $A[p..q-1]$, or 3) recurse on the subarray $A[q+1..r]$
This decision depends on where the \( i \)th smallest element falls relative to \( A[q] \).

Assuming \( T(n) \) to be monotonically increasing, we can upper-bound the time needed for a recursive call by that on largest possible input.

I.e., to obtain an upper bound, we assume that the \( i \)th element is always on the side of the partition with the greater number of elements.

For a given call of \textsc{Randomized-Select}, the indicator random variable \( X_k \) has value 1 for exactly one value of \( k \), and it is 0 for all other \( k \).

When \( X_k = 1 \), the two subarrays on which we might recurse have sizes \( k - 1 \) and \( n - k \).

We have the recurrence
\[
T(n) \leq \sum_{i=1}^{n} X_k \cdot (T(\text{max}(k - 1, n - k)) + O(n))
\]
\[
= \sum_{i=1}^{n} X_k \cdot T(\text{max}(k - 1, n - k)) + O(n)
\]

Taking expected values, we have
\[
E[T(n)] \leq E \left[ \sum_{i=1}^{n} X_k \cdot T(\text{max}(k - 1, n - k)) + O(n) \right]
\]
\[
= \sum_{i=1}^{n} E[X_k \cdot T(\text{max}(k - 1, n - k))] + O(n)
\]
\[
= \sum_{i=1}^{n} E[X_k] \cdot E[T(\max(k-1, n-k))] + O(n)
\]

\[
= \sum_{i=1}^{n} \frac{1}{n} \cdot E[T(\max(k-1, n-k))] + O(n)
\]

- The first Eq. on this slide follows by independence of random variables \(X_k\) and \(\max(k-1, n-k)\)
- Consider the expression
  \[
  \max(k-1, n-k) = \begin{cases} 
k - 1 & \text{if } k > \lfloor n/2 \rfloor 
n - k & \text{if } k \leq \lfloor n/2 \rfloor 
\end{cases}
\]

- If \(n\) is even, each term from \(T(\lfloor n/2 \rfloor)\) up to \(T(n-1)\) appears exactly twice in the summation, and if \(n\) is odd, all these terms appear twice and \(T(\lfloor n/2 \rfloor)\) appears once
- Thus, we have
  \[
  E[T(n)] \leq \frac{2}{n} \sum_{k=[n/2]}^{n-1} E[T(k)] + O(n)
  \]
- We can show that \(E[T(n)] = O(n)\) by substitution
- In summary, we can find any order statistic, and in particular the median, in expected linear time, assuming that the elements are distinct
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 $x$ 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 $O(\lg n)$
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 $T[h(x.\text{key})]$

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 $T[h(x.\text{key})]$

- 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 $T[h(x.\text{key})]$
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 $E[n_j] = \alpha = n/m$
**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

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

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

$$= \frac{1}{n} \sum_{i=1}^{n} \left( 1 + \sum_{j=i+1}^{n} \frac{1}{m} \right)$$
\[
\begin{align*}
&= 1 + \frac{1}{nm} \sum_{i=1}^{n} (n - i) \\
&= 1 + \frac{1}{nm} \left( \sum_{i=1}^{n} n - \sum_{i=1}^{n} i \right) \\
&= 1 + \frac{1}{nm} \left( n^2 - \frac{n(n + 1)}{2} \right) \\
&= 1 + \frac{n - 1}{2m} = 1 + \frac{\alpha}{2} + \frac{\alpha}{2n} \\
\end{align*}
\]
Thus, the total time required for a successful search is \( \Theta(2 + \alpha/2 + \alpha/2n) = \Theta(1 + \alpha) \).
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 $(112 \cdot 128) + 116 = 14452$. 
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 ($2^9 = 512$ and $2^{10} = 1024$).

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
\[ A \approx \left(\sqrt{5} - 1\right)/2 = 0.6180339887 \ldots \]
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 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

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

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

1. $i \leftarrow 0$
2. repeat
3. $j \leftarrow h(k, i)$
4. if $T[j] = k$
5. return $j$
6. $i \leftarrow i + 1$
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
We assume **uniform hashing** (UH):
- the probe sequence of each key is equally likely to be any of the \( m! \) permutations of \( \{0,1,\ldots,m-1\} \)
- UH generalizes the notion of SUH that produces not just a single number, but a whole probe sequence
- True uniform hashing is difficult to implement, however, and in practice suitable approximations (such as double hashing, defined below) are used

We examine three common techniques to compute the probe sequences required for open addressing: 1) linear probing, 2) quadratic probing, and 3) double hashing
- These techniques all guarantee that \( \langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle \) is a permutation of \( \langle 0,1,\ldots,m-1 \rangle \) for each key \( k \)
- No technique fulfills the assumption of UH;
  - none of them is capable of generating more than \( m^2 \) different probe sequences
- Double hashing has the greatest number of probe sequences and gives the best results
Linear probing

- Given a hash function $h': U \to \{0, 1, \ldots, m - 1\}$, an auxiliary hash function, use the function $h(k, i) = (h'(k) + i) \mod m$
- Given key $k$, we first probe the slot given by the auxiliary hash function $T[h'(k)]$
- We next probe slots $T[h'(k) + 1], \ldots, T[m - 1]$
- Wrap around to $T[0], T[1], \ldots, T[h'(k) - 1]$
- Initial probe determines the entire probe sequence, there are only $m$ distinct seq's

Linear probing is easy to implement, but it suffers from a problem known as primary clustering
- Long runs of occupied slots build up, increasing the average search time
- Clusters arise because an empty slot preceded by $i$ full slots gets filled next with probability $(i + 1)/m$
- Long runs of occupied slots tend to get longer, and the average search time increases
Quadratic probing

- Use a hash function of the form
  \[ h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m \]
  where \( c_1, c_2 \) are positive auxiliary constants
- Initial position probed is \( T[h'(k)] \); later positions are offset by amounts that depend in a quadratic manner on the probe number \( i \)
- This works much better than linear probing, but to make full use of the hash table, the values of \( c_1, c_2 \), and \( m \) are constrained

- Also, if two keys have the same initial probe position, then their probe sequences are the same, since \( h(k_1, 0) = h(k_2, 0) \) implies \( h(k_1, i) = h(k_2, i) \)
- This property leads to a milder form of clustering, called secondary clustering
- As in linear probing, the initial probe determines the entire sequence, and so only \( m \) distinct probe sequences are used
Double hashing

- One of the best methods available for open addressing, the permutations produced have many characteristics of random ones.
- Uses a hash function of the form
  \[ h(k, i) = (h_1(k) + ih_2(k)) \mod m \]
  where \( h_1 \) and \( h_2 \) are auxiliary hash functions.
- The initial probe goes to position \( T[h_1(k)] \); successive probes are offset from previous positions by the amount \( h_2(k) \), modulo \( m \).

Here we have a hash table of size 13 with \( h_1(k) = k \mod 13 \) and \( h_2(k) = 1 + (k \mod 11) \).

Since \( 14 \equiv 1 \mod 13 \) and \( 14 \equiv 3 \mod 11 \), we insert the key 14 into empty slot \( h_1(k) + 2h_2(k) = 9 \), after examining slots \( h_1(k) = 1 \) and \( h_1(k) + h_2(k) = 5 \) and finding them to be occupied.
The value $h_2(k)$ must be relatively prime to $m$ for the entire hash table to be searched.

A convenient way to ensure this condition is to let $m$ be a power of 2 and to design $h_2$ so that it always produces an odd number.

Another way is to let $m$ be prime and to design $h_2$ so that it always returns a positive integer less than $m$.

For example, we could choose $m$ prime and let $h_1(k) = k \mod m$, $h_2(k) = 1 + (k \mod m')$, where $m'$ is slightly less than $m$ (say, $m - 1$).

E.g., if $k = 123,456$, $m = 701$, and $m' = 700$, we have $h_1(k) = 80$ and $h_2(k) = 257$.

We first probe position 80, and then we examine every 257th slot (modulo $m$) until we find the key or have examined every slot.

When $m$ is prime or a power of 2, double hashing improves over linear or quadratic probing in that $\Theta(m^2)$ probe sequences are used, rather than $\Theta(m)$.

Each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence.