We assume **uniform hashing** (UH):

– the probe sequence of each key is equally likely to be any of the $m!$ permutations of $\langle 0,1,\ldots,m-1 \rangle$

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: linear probing, quadratic probing, and 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 \( h': U \rightarrow \{0,1,\ldots, m-1\} \), 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 \textit{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
  \[ h(k, i) = (h_1(k) + i h_2(k)) \mod m \]
  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 \( \equiv 1 \pmod{13} \) and 14 \( \equiv 3 \pmod{11} \), we insert the key 14 into empty slot 9, after examining slots 1 and 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.
Analysis of open-address hashing

- Let us express our analysis of in terms of the load factor \( \alpha = n/m \) of the hash table.
- Now at most one element occupies each slot, and thus \( n \leq m \), which implies \( \alpha \leq 1 \).
- Assume that we are using uniform hashing.
- In this idealized scheme, the probe sequence \( \langle h(k, 0), h(k, 1), \ldots, h(k, m - 1) \rangle \) used to insert or search for each key \( k \) is equally likely to be any permutation of \( \langle 0, 1, \ldots, m - 1 \rangle \).

Theorem 11.6  Given an open-address hash table with \( \alpha = n/m < 1 \), the expected number of probes in an unsuccessful search is at most \( 1/(1 - \alpha) \), assuming UH.

Proof  Every probe but the last accesses an occupied slot that does not contain the desired key, and the last slot probed is empty.
Define the random variable \( X \) to be the number of probes made in an unsuccessful search, and also define the event \( A_i, i = 1, 2, \ldots \), to be the event that an \( i \)th probe occurs and it is to an occupied slot.
The event \( \{ X \geq i \} \) is the intersection of events 
\[ A_1 \cap A_2 \cap \cdots \cap A_{i-1} . \]

Bound \( \Pr \{ X \geq i \} \) by \( \Pr \{ A_1 \cap A_2 \cap \cdots \cap A_{i-1} \} \) which by Exercise C.2-5
\[
\Pr \{ A_{i-1} | A_1 \cap A_2 \cap \cdots \cap A_{i-2} \} = \Pr \{ A_1 \} \cdot \Pr \{ A_2 | A_1 \} \cdot \Pr \{ A_3 | A_1 \cap A_2 \} \cdots
\]

There are \( n \) elements and \( m \) slots, so
\[ \Pr \{ A_1 \} = \frac{n}{m} . \]

For \( j > 1 \), the probability that there is a \( j \)th probe and it is to an occupied slot, given that the first 
\( j - 1 \) probes were to occupied slots, is \( \frac{(n - j + 1)}{(m - j + 1)} \).

This probability follows because we would be finding one of the remaining \( (n - j + 1) \) elements 
in one of the \( (m - j + 1) \) unexamined slots, and by the assumption of UH, the probability is the ratio of 
these quantities.

\( n < m \) implies that \( (n - j) / (m - j) \) \( \leq \frac{n}{m} \) for all 
\[ 0 \leq j < m . \]

Therefore, we have for all \( 1 \leq i \leq m \),
\[
\Pr \{ X \geq i \} = \frac{n}{m} \cdot \frac{n - 1}{m - 1} \cdots \frac{n - i + 2}{m - i + 2} 
\leq \left( \frac{n}{m} \right)^{i-1} = a^{i-1}
\]
• Now, because $E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$

$E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$

$\leq \sum_{i=1}^{\infty} \alpha^{i-1}$

$= \sum_{i=0}^{\infty} \alpha^{i}$

$= \frac{1}{1-\alpha}$

• This bound of $\frac{1}{(1-\alpha)} = 1 + \alpha + \alpha^2 + \cdots$ has an intuitive interpretation
  – We always make the first probe
  – With probability approximately $\alpha$, it finds an occupied slot, so that we need to probe again
  – With probability approx. $\alpha^2$, the first two slots are occupied and we make a third probe, …

• If $\alpha$ is a constant, Theorem 11.6 predicts that an unsuccessful search runs in $O(1)$ time

• If the table is half full, the avg. number of probes in an unsuccessful search is $\leq \frac{1}{(1-.5)} = 2$

• If it is 90% full, the average number of probes is $\leq \frac{1}{(1-.9)} = 10$
**Corollary 11.7** Inserting an element into an open-address hash table with load factor $\alpha$ requires at most $1/(1 - \alpha)$ probes on average, assuming uniform hashing.

**Proof** An element is inserted only if there is room in the table, and thus $\alpha < 1$. Inserting a key requires an unsuccessful search followed by placing the key into the first empty slot found. Thus, the expected number of probes is at most $1/(1 - \alpha)$.

---

**Theorem 11.8** Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$$

assuming UH and assuming that each key in the table is equally likely to be searched for.
12 Binary Search Trees

- The keys in a binary search tree (BST) are always stored in such a way as to satisfy the binary-search-tree property:
  - Let $x$ be a node in a BST
    - If $y$ is a node in the left subtree of $x$, then $y.key \leq x.key$
    - If $y$ is a node in the right subtree of $x$, then $y.key \geq x.key$
The BST property allows us to print out all the keys in a BST in sorted order by a simple recursive algorithm, called an *inorder tree walk*

- It prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree
- *A preorder tree walk* prints the root before the values in either subtree
- *A postorder tree walk* prints the root after the values in its subtrees

**INORDER-TREE-WALK** \(x\)

1. *if* \(x \neq \text{NIL}\)
2. **INORDER-TREE-WALK** \(x.\text{left}\)
3. print \(x.\text{key}\)
4. **INORDER-TREE-WALK** \(x.\text{right}\)

- It takes \(\Theta(n)\) time to walk an \(n\)-node BST
  - after the initial call, the procedure calls itself recursively exactly twice for each node
  - once for its left child and once for its right child
12.2 Querying a binary search tree

- In addition to the SEARCH operation, BSTs can support queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR.
- We show how to support each operation in time $O(h)$ on any BST of height $h$.
- Given a pointer to the root of the tree and a key $k$, TREE-SEARCH returns a pointer to a node with key $k$ if one exists; otherwise, it returns NIL.

\[
\text{TREE-SEARCH}(x, k)
\]
1. if $x == \text{NIL}$ or $k == x.\text{key}$
2. return $x$
3. if $k < x.\text{key}$
4. return TREE-SEARCH($x.\text{left}, k$)
5. else return TREE-SEARCH($x.\text{right}, k$)
• We can rewrite this in an iterative fashion by “unrolling” the recursion into a while loop
• On most computers, the iterative version is more efficient

**ITERATIVE-TREE-SEARCH**(x, k)
1. while x ≠ NIL and k ≠ x.key
2. if ₦ < x.key
3. x = x.left
4. else x = x.right
5. return x

---

**Minimum and maximum**

• We find a minimum element by following left child pointers until we encounter a NIL

**TREE-MINIMUM**(x)
1. while x.left ≠ NIL
2. x = x.left
3. return x
The BST property guarantees that TREE-MINIMUM is correct.
The pseudocode for TREE-MAXIMUM is symmetric.
Both of these procedures run in $O(h)$ time on a tree of height $h$ since, as in TREE-SEARCH, the sequence of nodes encountered forms a simple path downward from the root.

Successor and predecessor

The structure of a BST lets us determine the successor of a node without comparing keys.

TREE-SUCCESSOR($x$)

1. if $x.right \neq NIL$
2. return TREE-MINIMUM($x.right$)
3. $y = x.p$
4. while $y \neq NIL$ and $x == y.right$
5. $x = y$
6. $y = y.p$
7. return $y$
• The running time of TREE-SUCCESSOR on a tree of height \( h \) is \( O(h) \), we either follow a simple path up down the tree
• The procedure TREE-PREDECESSOR also runs in time \( O(h) \)

**Theorem 12.2** We can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR so that each one runs in \( O(h) \) time on a BST of height \( h \)

---

12.3 Insertion and deletion

**TREE-INSERT** \((T, z)\)

1. \( y = \text{NIL} \)
2. \( x = T.\text{root} \)
3. while \( x \neq \text{NIL} \)
4. \( y = x \)
5. if \( z.\text{key} < x.\text{key} \)
6. \( x = x.\text{left} \)
7. else \( x = x.\text{right} \)
8. \( z.p = y \)
9. if \( y == \text{NIL} \)
10. \( T.\text{root} = z \)

// tree \( T \) was empty
11. elseif \( z.\text{key} < y.\text{key} \)
12. \( y.\text{left} = z \)
13. else \( y.\text{right} = z \)

• TREE-INSERT runs in \( O(h) \) time on a tree of height \( h \)
Deleting a node $z$ from a BST $T$ has three basic cases but one of them is a bit tricky

- $z$ has 0 children: remove it by modifying its parent to replace $z$ with \texttt{NIL}.
- $z$ has just 1 child: elevate that child to take $z$’s position in the tree by modifying $z$’s parent to replace $z$ by $z$’s child.
- $z$ has 2 children: find $z$’s successor $y$ — in $z$’s right subtree — and have $y$ take $z$’s position. The rest of $z$’s original right subtree becomes $y$’s new right subtree, and $z$’s left subtree becomes $y$’s new left subtree.
The last case is the tricky one because it matters whether $y$ is $z$’s right child.

If $z$ has no left child, then we replace $z$ by its right child, which may or may not be NIL.

If $z$ has just one child, which is its left child, then we replace $z$ by it.

Otherwise, $z$ has both children. $z$’s successor $y$ lies in right subtree and has no left child. We want to replace $z$ with $y$ in the tree.

- If $y$ is $z$’s right child, then we replace $z$ by $y$, leaving $y$’s right child alone.

- Otherwise, $y$ lies within $z$’s right subtree but is not $z$’s right child. In this case, we first replace $y$ by its own right child, and then we replace $z$ by $y$. 
• Subroutine \textsc{Transplant} replaces the subtree rooted at $u$ with the subtree rooted at $v$, $u$’s parent becomes $v$’s parent, and $u$’s parent ends up having $v$ as its appropriate child.

\textsc{Transplant}(T,u,v)

1. if $u.p == \text{NIL}$
2. $T\text{.root} = v$
3. elseif $u == u.p\text{.left}$
4. $u.p\text{.left} = v$
5. else $u.p\text{.right} = v$
6. if $v \neq \text{NIL}$
7. $v.p = u.p$

\begin{figure}[h]
\includegraphics[width=\textwidth]{transplant_diagram.png}
\end{figure}
TREE-DELETE \((T, z)\)

1. if \(z\text{.left} == \text{NIL}\)
2. \text{TRANSPLANT}(T, z, z\text{.right})
3. elseif \(z\text{.right} == \text{NIL}\)
4. \text{TRANSPLANT}(T, z, z\text{.left})
5. else \(y = \text{TREE-MINIMUM}(z\text{.right})\)
6. if \(y\text{.p} \neq z\)
7. \text{TRANSPLANT}(T, y, y\text{.right})
8. \(y\text{.right} = z\text{.right}\)
9. \(y\text{.right.p} = y\)
10. \text{TRANSPLANT}(T, z, y)
11. \(y\text{.left} = z\text{.left}\)
12. \(y\text{.left.p} = y\)

- Each line of TREE-DELETE, including the calls to TRANSPLANT, takes constant time, except for the call to TREE-MINIMUM
- TREE-DELETE runs in \(O(h)\) time on a tree of height \(h\)

**Theorem 12.3** We can implement the dynamic-set operations INSERT and DELETE so that each one runs in \(O(h)\) time on a binary search tree of height \(h\)
13 Red-Black Trees

- A red-black tree (RBT) is a BST with one extra bit of storage per node: color, either RED or BLACK
- Constraining the node colors on any path from the root to a leaf
  - Ensures that no such path is more than twice as long as any other, so that the tree is approximately balanced

A red-black tree is a binary tree that satisfies the following red-black properties:
1. Every node is either RED or BLACK
2. The root is BLACK
3. Every leaf (NIL) is BLACK
4. If a node is RED, then both its children are BLACK
5. For each node, all simple paths from the node to descendant leaves contain the same number of BLACK nodes
We call the number of black nodes on any simple path from a node \( x \) (not inclusive) down to a leaf the **black-height** of the node, denoted \( bh(x) \).

- By property 5, black-height is well defined, since all descending simple paths from the node have the same number of black nodes.
- We define the black-height of a red-black tree to be the black-height of its root.
Lemma 13.1 A red-black tree with \( n \) internal nodes has height at most \( 2\lg(n + 1) \).

**Proof** Use induction (on the height of \( x \)) to show that the subtree rooted at any node \( x \) contains at least \( 2^{bh(x)} - 1 \) internal nodes.

- If the height of \( x \) is 0, then \( x \) must be a leaf (\( T.nil \)), and the subtree rooted at \( x \) indeed contains at least \( 2^{bh(x)} - 1 = 2^0 - 1 = 0 \) internal nodes.
- Consider a node \( x \) that has positive height and is an internal node with two children.

- Each child has a black-height of either \( bh(x) \) or \( bh(x) - 1 \), depending on whether its color is red or black, respectively.
- The height of a child of \( x \) < the height of \( x \) itself; by the inductive hypothesis each child has at least \( 2^{bh(x) - 1} - 1 \) internal nodes.
- Thus, the subtree rooted at \( x \) contains at least \( 2^{bh(x)} - 1 + 1 = 2^{bh(x)} - 1 \) internal nodes, which proves the claim.

To complete the proof of the lemma, let \( h \) be the height of the tree.
According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least $h/2$; thus,

$$n \geq 2^{h/2} - 1.$$ 

Moving the 1 to the left-hand side and taking logarithms on both sides yields

$$\lg(n + 1) \geq h/2, \text{ or } h \leq 2\lg(n + 1).$$

As an immediate consequence, we can implement operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(\lg n)$ time on RBTs.

- TREE-INSERT and TREE-DELETE do not guarantee that the modified BST will be a red-black tree.
- We shall see how to support these two operations in $O(\lg n)$ time.