Trees

Important terminology:

- **root**
- **parent**
- **child**
- **leaf**
- **edge**

Some uses of trees:

- model arithmetic expressions and other expressions to be parsed
- model game-theory approaches to solving problems: nodes are configurations, children result from different moves
- a clever implementation of priority queue ADT
- search trees, each node holds a data item
Trees (cont’d)

Some more terms:

- **path**: sequence of edges, each edge starts with the node where the previous edge ends
- **length of path**: number of *edges* in it
- **height of a node**: length of *longest* path from the node to a leaf
- **height of tree**: height of root
- **depth (or level) of a node**: length of path from root to the node
- **depth of tree**: maximum depth of any leaf

**Fact**: The depth of a tree equals the height of the tree.
Binary Trees

**Binary tree:** a tree in which each node has at most two children.

**Complete binary tree:** tree in which all leaves are on the same level and each non-leaf node has exactly two children.

**Important Facts:**

- A complete binary tree with $L$ levels contains $2^L - 1$ nodes.
- A complete binary tree with $n$ nodes has approximately $\log_2 n$ levels.
Binary Trees (cont’d)

**Leftmost binary tree:** like a complete binary tree, except that the bottom level might not be completely filled in; however, all leaves at bottom level are as far to the left as possible.

![Binary Trees Diagram](image)

**Important Facts:**

- A leftmost binary tree with $L$ levels contains between $2^{L-1}$ and $2^L - 1$ nodes.

- A leftmost binary tree with $n$ nodes has approximately $\log_2 n$ levels.
Binary Heap

Now suppose that there is a data item, called a key, inside each node of a tree.

A **binary heap** (or min-heap) is a

- leftmost binary tree with keys in the nodes that
- satisfies the **heap ordering property**: every node has a key that is less than or equal to the keys of all its children.

*Do not confuse this use of “heap” with its usage in memory management!*

**Important Fact:** The same set of keys can be organized in many different heaps. There is no required order between siblings’ keys.
Using a Heap to Implement a Priority Queue

To implement the priority queue operation $$\text{insert}(x)$$:
1. Make a new node in the tree in the next available location, marching across from left to right.
2. Put $$x$$ in the new node.
3. “Bubble $$x$$ up” the tree until finding a correct place:
   - if $$x < \text{parent’s key}$$, then swap keys and continue.
Time: $$O(\log n)$$

To implement the priority queue operation $$\text{remove}()$$:
Tricky part is how to remove the root without messing up the tree structure.
1. Swap the key in the root with the key (call it $$y$$) in the rightmost node on the bottom level.
2. Remove the rightmost node on the bottom level.
3. “Bubble down” the new root’s key $$y$$ until finding a correct place:
   - if $$y > \text{at least one child’s key}$$, then swap $$y$$ with smallest child’s key and continue.
Time: $$O(\log n)$$. 
Using a Heap to Implement a PQ (cont’d)

<table>
<thead>
<tr>
<th>PQ operation</th>
<th>sorted array or linked list</th>
<th>unsorted array or linked list</th>
<th>heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>remove (min)</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

No longer have the severe tradeoffs of the array and linked list representations of priority queue. By keeping the keys “semi-sorted” instead of fully sorted, we can decrease the tradeoff between the costs of insert and remove.
Heap Sort

Recall the sorting algorithm that used a priority queue:

1. insert the elements to be sorted, one by one, into a priority queue.

2. remove the elements, one by one, from the priority queue; they will come out in sorted order.

If the priority queue is implemented with a heap, the running time is $O(n \log n)$.

This is much better than $O(n^2)$.

This algorithm is called heap sort.
Linked Structure Implementation of Heap

To implement a heap with a linked structure, each node of the tree will be represented with an object containing

- key (data)
- pointer to parent node
- pointer to left child
- pointer to right child

To find the next available location for insert, or the rightmost node on the bottom level for remove, in constant time, keep all nodes on the same level in a linked list. Thus each node will also have

- pointer to left neighbor on same level
- pointer to right neighbor on same level

Then keep a single pointer for the whole heap that points to the rightmost node on the bottom level.
Array Implementation of Heap

Fortunately, there’s a nifty way to implement a heap using an array, based on an interesting observation: If you number the nodes in a leftmost binary tree, starting at the root and going across levels and down levels, you see a pattern:

- Node number $i$ has left child $2 \cdot i$.
- Node number $i$ has right child $2 \cdot i + 1$.
- If $2 \cdot i > n$, then $i$ has no left child.
- If $2 \cdot i + 1 > n$, then $i$ has no right child.
- Therefore, node number $i$ is a leaf if $2 \cdot i > n$.
- The parent of node $i$ is $\lfloor i/2 \rfloor$, as long as $i > 1$.
- Next available location for insert is index $n + 1$.
- Rightmost node on the bottom level is index $n$. 
Array Implementation of Heap (cont’d)

Representation consists of

- array $A[1..\text{max}]$ (ignore location 0)
- integer $n$, which is initially 0, holding number of elements in heap

To implement $\text{insert}(x)$ (ignoring overflow):

```plaintext
n := n+1     // make a new leaf node
A[n] := x     // new node’s key is initially x
cur := n      // start bubbling x up
parent := cur/2
    // current node is not the root and its key
    // has not found final resting place
    swap A[cur] and A[parent]
cur := parent     // move up a level in the tree
parent := cur/2
endwhile
```
Array Implementation of Heap (cont’d)

To implement **remove** (ignoring underflow):

minKey := A[1] // smallest key, to be returned
  //       rightmost leaf on bottom level
n := n-1 // delete rightmost leaf on bottom level
cur := 1 // start bubbling down key in root
Lchild := 2*cur
Rchild := 2*cur + 1
while (Lchild <= n) && (A[minChild()] < A[cur]) do
  // current node is not a leaf and its key has
  // not found final resting place
  swap A[cur] and A[minChild()]
  cur := minChild() // move down a level in the tree
  Lchild := 2*cur
  Rchild := 2*cur + 1
endwhile
return minKey

minChild(): // returns index of child w/ smaller key
  min := Lchild
  if (Rchild <= n) && (A[Rchild] < A[Lchild]) then
    // node has a right child and it is smaller
    min := RChild
  endif
  return min
Binary Tree Traversals

Now consider any kind of binary tree with data in the nodes, not just leftmost binary trees.

In many applications, we need to traverse a tree: “visit” each node exactly once. When the node is visited, some computation can take place, such as printing the key.

There are three popular kinds of traversals, differing in the order in which each node is visited in relation to the order in which its left and right subtrees are visited:

- **inorder traversal**: visit the node IN between visiting the left subtree and visiting the right subtree
- **preorder traversal**: visit the node BEFORE visiting either subtree
- **postorder traversal**: visit the node AFTER visiting both its subtrees

*In all cases, it is assumed that the left subtree is visited before the right subtree.*
Binary Tree Traversals (cont’d)

preorder(x):
  if x is not empty then
    visit x
    preorder(leftchild(x))
    preorder(rightchild(x))

inorder(x):
  if x is not empty then
    inorder(leftchild(x))
    visit x
    inorder(rightchild(x))

postorder(x):
  if x is not empty then
    postorder(leftchild(x))
    postorder(rightchild(x))
    visit x

- preorder:  a b d e c f g h i
- inorder:  d e b a f c h g i
- postorder:  e d b f h i g c
Binary Tree Traversals (cont’d)

These traversals are particularly interesting when the binary tree is a parse tree for an arithmetic expression:

- Postorder traversal results in the postfix representation of the expression.
- Preorder gives prefix representation.
- Does inorder give infix? No, because there are no parentheses to indicate precedence.

```
1 + 2 3 *
```

- preorder:  * + 5 3 - 2 1
- inorder:  5 3 + 2 1 - *
- postorder:  5 + 3 * 2 - 1
Representation of a Binary Tree (cont’d)

class Tree {
    TreeNode root;
    // other information...

    void preorderTraversal() {
        preorder(root);
    }

    preorder(TreeNode t) {
        if (t != null) { // stopping case for recursion
            t.visit(); // user-defined visit method
            preorder(t.left);
            preorder(t.right);
        }
    }
}

But we haven’t yet talked about how you actually MAKE a binary tree. We’ll do that next, when we talk about binary SEARCH trees.
Dictionary ADT Specification

So far, we’ve seen the abstract data types

- priority queue, with operations insert, remove (min),...
- stack, with operations push, pop,...
- queue, with operations enq, deq,...
- list, with operations insert, delete, replace,...

Another useful ADT is a dictionary (or table). The abstract state of a dictionary is a set of elements, each of which has a key. The main operations are:

- insert an element
- delete an arbitrary element (not necessarily the highest priority one)
- search for a particular element

Some additional operations are:

- find the minimum element,
- find the maximum element,
- print out all elements in sorted order.
Dictionary ADT Applications

The dictionary (or table) ADT is useful in “database” type applications.

For instance, student records at a university can be kept in a dictionary data structure:

- When a new student enrolls, an insert is done.
- When a student graduates, a delete is done.
- When information about a student needs to be updated, a search is done, using either the name or ID number as the key.
- Once the search has located the record for that student, the data can be updated.
- When information about student needs to be retrieved, a search is done.

The world is full of information databases, many of them extremely large (imagine what the IRS has).

*When the number of elements gets very large, efficient implementations of the dictionary ADT are essential.*
Dictionary Implementations

We will study a number of implementations:

Search Trees

- (basic) binary search trees
- balanced search trees
  - AVL trees (binary)
  - red-black trees (binary)
  - B-trees (not binary)
- tries (not binary)

Hash Tables

- open addressing
- chaining
Binary Search Tree

Recall the *heap ordering property* for binary heaps: each node’s key is smaller than the keys in both children.

*Another* ordering property is the *binary search tree property*: for each node $x$,

- all keys in the left subtree of $x$ are less than the key in $x$, and
- all keys in the right subtree of $x$ are greater than the key in $x$.

A *binary search tree (BST)* is a binary tree that satisfies the binary search tree property.
Searching in a BST

To search for a particular key in a binary search tree, we take advantage of the binary search tree property:

```plaintext
search(x,k): // x is node where search starts
-------------- // k is key searched for
if x is null then // stopping case for recursion
    return "not found"
else if k = the key of x then
    return x
else if k < the key of x then
    search(leftchild(x),k) // recursive call
else // k > the key of x
    search(rightchild(x),k) // recursive call
endif
```

The top level call has x equal to the root of the tree.

In the previous tree, the search path for 17 is 19, 10, 16, 17, and the search path for 21 is 19, 22, 20, null.

**Running Time:** \(O(d)\), where \(d\) is depth of tree. If BST is a chain, then \(d = n - 1\).
Searching in a BST (cont’d)

Iterative version of search:

search(x,k):
-------------
while x != null do
    if k = the key of x then
        return x
    else if k < the key of x then
        x := leftchild(x)
    else // k > the key of x
        x := rightchild(x)
    endif
endwhile
return "not found"

As in the recursive version, you keep going down the tree until you either find the key or hit a leaf.

The comparison of the search key with the node key tells you at each level whether to continue the search to the left or to the right.

**Running Time:** \( O(d) = O(n). \)
Searching in a Balanced BST

If the tree is a complete binary tree, then the depth is $O(\log n)$, and thus the search time is $O(\log n)$.

Binary trees with $O(\log n)$ depth are considered balanced: there is balance between the number of nodes in the left subtree and the number of nodes in the right subtree of each node.

You can have binary trees that are approximately balanced, so that the depth is still $O(\log n)$, but might have a larger constant hidden in the big-oh.

As an aside, a binary heap does not have an efficient search operation: Since nodes at the same level of the heap have no particular ordering relationship to each other, you will need to search the entire heap in the worst case, which will be $O(n)$ time, even though the tree is perfectly balanced and only has depth $O(\log n)$. 
Inserting into a BST

To insert a key \( k \) into a binary search tree, search starting at the root until finding the place where the key should go. Then link in a node containing the key.

\[
\text{insert}(x,k):
\]

\[
\begin{align*}
\text{if } x = \text{null} & \text{ then} \\
& \text{make a new node containing } k \\
& \text{return new node} \\
\text{else if } k = \text{the key of } x & \text{ then} \\
& \text{return null } // \text{ key already exists} \\
\text{else if } k < \text{the key of } x & \text{ then} \\
& \text{leftchild}(x) := \text{insert}(\text{leftchild}(x),k) \\
& \text{return } x \\
\text{else } // k > \text{the key of } x & \text{ then} \\
& \text{rightchild}(x) := \text{insert}(\text{rightchild}(x),k) \\
& \text{return } x
\end{align*}
\]

Insert called on node \( x \) returns node \( x \), unless \( x \) is null, in which case insert returns a new node. As a result, a child of a node is only changed if it is originally non-existent.

**Running Time:** \( O(d) = O(n) \).
Inserting into a BST (cont’d)

after inserting 2, then 18, then 21:

```plaintext
  19
 /   \\  
10    22
 / \\
 4  16 20
 / \  / \\ 
13 17 20 26
   / \\  
 18 21 27
```
Finding Min and Max in Binary Search Tree

**Fact:** The smallest key in a binary tree is found by following left children as far as you can.

![Binary Search Tree Diagram]

**Running Time:** $O(d) = O(n)$.

Guess how to find the largest key and how long it takes.

Min is 4 and max is 27.
Printing a BST in Sorted Order

Cute tie-in between tree traversals and BST’s.

**Theorem:** Inorder traversal of a binary search tree visits the nodes in sorted order of the keys.

Inorder traversal on previous tree gives: 4, 10, 13, 16, 17, 19, 20, 22, 26, 27.

**Proof:** Let’s look at some small cases and then use induction for the general case.

*Case 1:* Single node. Obvious.

*Case 2:* Two nodes. Check the two cases.

*Case n:* Suppose true for trees of size 1 through $n-1$. Consider a tree of size $n$: 
Printing a BST in Sorted Order (cont’d)

$L$ contains at most $n - 1$ keys, and $R$ contains at most $n - 1$ keys.

Inorder traversal:

- prints out all keys in $R$ in sorted order, by induction.
- then prints out key in $r$, which is larger than all keys in $R$,
- then prints out all keys in $L$ in sorted order, by induction. All these keys are greater than key in $r$.

Running Time: $O(n)$ since each node is handled once.
Tree Sort

Does previous theorem suggest yet another sorting algorithm to you?

**Tree Sort:** Insert all the keys into a BST, then do an inorder traversal of the tree.

**Running Time:** \( O(n^2) \), since each of the \( n \) inserts takes \( O(n) \) time.
Finding Successor in a BST

The successor of a node $x$ in a BST is the node whose key immediately follows $x$’s key, in sorted order.

*Case 1*: If $x$ has a right child, then the successor of $x$ is the minimum in the right subtree: follow $x$’s right pointer, then follow left pointers until there are no more.

```
Path to find successor of 19 is 19, 22, 20.
```
Finding Successor in a BST (cont’d)

**Case 2:** If \( x \) does not have a right child, then find the lowest ancestor of \( x \) whose left child is also an ancestor of \( x \). (I.e., follow parent pointers from \( x \) until reaching a key larger than \( x \)’s.)

Path to find successor of 17 is 17, 16, 10, 19.

If you never find an ancestor that is larger than \( x \)’s key, then \( x \) was already the maximum and has no successor.

Path to try to find successor of 27 is 27, 26, 22, 19.

**Running Time:** \( O(d) = O(n) \).
Finding Predecessor in a BST

The **predecessor** of a node \( x \) in a BST is the node whose key immediately precedes \( x \)'s key, in sorted order. To find it, do the “mirror image” of the algorithm for successor.

**Case 1**: If \( x \) has a left child, then the predecessor of \( x \) is the maximum in the left subtree: follow \( x \)'s left pointer, then follow right pointers until there are no more.

**Case 2**: If \( x \) does not have a left child, then find the lowest ancestor of \( x \) whose right child is also an ancestor of \( x \). (I.e., follow parent pointers from \( x \) until reaching a key smaller than \( x \)'s.)

If you never find an ancestor that is smaller than \( x \)'s key, then \( x \) was already the minimum and has no predecessor.

**Running Time**: \( O(d) = O(n) \).
Deleting a Node from a BST

Case 1: $x$ is a leaf. Then just delete $x$’s node from the tree.

Case 2: $x$ has only one child. Then “splice out” $x$ by connecting $x$’s parent directly to $x$’s (only) child.

Case 3: $x$ has two children. Use the same strategy as binary heap: Instead of removing the root node, choose another node that is easier to remove, and swap the data in the nodes.

1. Find the successor (or predecessor) of $x$, call it $y$. It is guaranteed to exist since $x$ has two children.

2. Delete $y$ from the tree. Since $y$ is the successor, it has no left child, and thus it can be dealt with using either Case 1 or Case 2.

3. Replace $x$’s key with $y$’s key.

Running Time: $O(d) = O(n)$. 
Deleting a Node from a BST (cont’d)

![BST Diagram]

after deleting 13, then 26, then 10:
Hash Table Implementation of Dictionary ADT

Another implementation of the Dictionary ADT is a hash table.

Hash tables support the operations

- insert an element
- delete an arbitrary element
- search for a particular element

with constant average time performance. This is a significant advantage over even balanced search trees, which have average times of $O(\log n)$.

The disadvantage of hash tables is that the operations min, max, pred, succ take $O(n)$ time; and printing all elements in sorted order takes $O(n \log n)$ time.
Main Idea of Hash Table

Main idea: exploit random access feature of arrays: the i-th entry of array A can be accessed in constant time, by calculating the address of A[i], which is offset from the starting address of A.

Simple example: Suppose all keys are in the range 0 to 99. Then store elements in an array A with 100 entries. Initialize all entries to some empty indicator.

- To insert x with key k: A[k] := x.
- To search for key k: check if A[k] is empty.
- To delete element with key k: A[k] := empty.

All times are $O(1)$.

But this idea does not scale well.
Hash Functions

Suppose

- elements are student records
- school has 40,000 students,
- keys are social security numbers (000-00-0000).

Since there are 1 billion possible SSN’s, we need an array of length 1 billion. And most of it will be wasted, since only \( \frac{40,000}{1,000,000,000} = \frac{1}{25,000} \) fraction is nonempty.

Instead, we need a way to *condense* the keys into a smaller range.

Let \( M \) be the size of the array we are willing to provide.

Use a **hash function**, \( h \), to convert each key to an array index. Then \( h \) maps key values to integers in the range 0 to \( M - 1 \).
Simple Hash Function Example

Suppose keys are integers. Let the hash function be $h(k) = k \mod M$. Notice that this always gives you something in the range 0 to $M - 1$ (an array index).

- To insert $x$ with key $k$: $A[h(k)] := x$
- To search for element with key $k$: check if $A[h(k)]$ is empty
- To delete element with key $k$: set $A[h(k)]$ to empty.

All times are $O(1)$, assuming the hash function can be computed in constant time.

| 0 | 1 | 2 | x | \ldots | 99 |

key is $k$ and $h(k) = 2$

The key to making this work is to choose hash function $h$ and table size $M$ properly (they interact).
Collisions

In reality, any hash function will have **collisions**: when two different keys hash to the same value:

\[ h(k_1) = h(k_2), \text{ although } k_1 \neq k_2. \]

This is inevitable, since the hash function is squashing down a large domain into a small range.

For example, if \( h(k) = k \mod M \), then \( k_1 = 0 \) and \( k_2 = M \) collide since they both hash to 0 (0 mod \( M \) is 0, and \( M \mod M \) is also 0).

What should you do when you have a collision? Two common solutions are

1. chaining, and
2. open addressing
Chaining

Keep all data items that hash to the same array location in a linked list:

- to insert element $x$ with key $k$: put $x$ at beginning of linked list at $A[h(k)]$
- to search for element with key $k$: scan the linked list at $A[h(k)]$ for an element with key $k$
- to delete element with key $k$: do search, if search is successful then remove element from the linked list

Worst case times, assuming computing $h$ is constant:

- insert: $O(1)$.
- search and delete: $O(n)$. Worst case is if all $n$ elements hash to same location.
Good Hash Functions for Chaining

*Intuition:* Hash function should spread out the data evenly among the $M$ entries in the table.

*More formally:* any key should be equally likely to hash to any of the $M$ locations.

Impractical to check in practice since the probability distribution on the keys is usually not known.

For example: Suppose the symbol table in a compiler is implemented with a hash table. The compiler writer cannot know in advance which variable names will appear in each program to be compiled.

Heuristics are used to approximate this condition: try something that seems reasonable, and run some experiments to see how it works.
Good Hash Functions for Chaining (cont’d)

Some issues to consider in choosing a hash function:

- Exploit application-specific information. For symbol table example, take into account the kinds of variables names that people often choose (e.g., x1). Try to avoid collisions on these names.

- Hash function should depend on all the information in the keys. For example: if the keys are English words, it is not a good idea to hash on the first letter, since many words begin with S and few with X.
Average Case Analysis of Chaining

Define **load factor** of hash table with $M$ entries and $n$ keys to be $\lambda = n/M$. (How full the table is.)

Assume a hash function that is ideal for chaining (any key is equally likely to hash to any of the $M$ locations).

**Fact:** Average length of each linked list is $n/M = \lambda$.

The **average** running time for chaining:

- **Insert:** $O(1)$ (same as worst case).
- **Unsuccessful Search:** $O(1 + \lambda)$. $O(1)$ time to compute $h(k)$; $\lambda$ items, on average, in the linked list are checked until discovering that $k$ is not present.
- **Successful Search:** $O(1 + \lambda/2)$. $O(1)$ time to compute $h(k)$; on average, key being sought is in middle of linked list, so $\lambda/2$ comparisons needed to find $k$.
- **Delete:** Essentially the same as search.

For these times to be $O(1)$, $\lambda$ must be $O(1)$, so $n$ cannot be too much larger than $M$. 
Open Addressing

With this scheme, there are no linked lists. Instead, all elements are stored in the table proper.

If there is a collision, you have to probe the table – check whether a slot (table entry) is empty – repeatedly until finding an empty slot.

You must pick a pattern that you will use to probe the table.

The simplest pattern is to start at \( h(k) \) and then check \( h(k) + 1, h(k) + 2, h(k) + 3, \ldots \), wrapping around to check 0, 1, 2, etc. if necessary, until finding an empty slot. This is called linear probing.

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
F & F & F & F & F & F & F & F & F \\
\end{array}
\]

If \( h(k) = 7 \), the probe sequence will be 7, 8, 0, 1, 2, 3. (F means full.)
Clustering

A problem with linear probing: clusters can build up. A cluster is a contiguous group of full slots. If an insert probe sequence begins in a cluster,

- it takes a while to get out of the cluster to find an empty slot,

- then inserting the new element just makes the cluster even bigger.

To reduce clustering, change the probe increment to skip over some locations, so locations are not checked in linear order.

There are various schemes for how to choose the increments; in fact, the increment to use can be a function of how many probes you have already done.
Clustering (cont’d)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

If the probe sequence starts at 7 and the probe increment is 4, then the probe sequence will be 7, 2, 6.

*Warning!* The probe increment must be relatively prime to the table size (meaning that they have no common factors): otherwise you will not search all locations.

For example, suppose you have table size 9 and increment 3. You will only search 1/3 of the table locations.
Double Hashing

Even when “non-linear” probing is used, it is still true that two keys that hash to the same location will follow the same probe sequence.

To get around this problem, use **double hashing**:

1. One hash function, $h_1$, is used to determine where to start probing.

2. A second hash function, $h_2$, is used to determine the probe sequence.

If the hash functions are chosen properly, different keys that have the same starting place will have different probe increments.
Double Hashing Example

Let \( h_1(k) = k \mod 13 \) and \( h_2(k) = 1 + (k \mod 11) \).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>79</td>
<td>69</td>
<td>98</td>
<td>72</td>
<td>15</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- To insert 14: start probing at \( 14 \mod 13 = 1 \). Probe increment is \( 1 + (14 \mod 11) = 4 \). Probe sequence is \( 1, 5, 9, 0 \).
- To insert 27: start probing at \( 27 \mod 13 = 1 \). Probe increment is \( 1 + (27 \mod 11) = 6 \). Probe sequence is \( 1, 7, 0, 6 \).
- To search for 18: start probing at \( 18 \mod 13 = 5 \). Probe increment is \( 1 + (18 \mod 11) = 8 \). Probe sequence is \( 5, 0 \) – not in table.
Deleting with Open Addressing

Open addressing has another complication:

- to insert: probe until finding an empty slot.
- to search: probe until finding the key being sought or an empty slot (which means not there)

Suppose we use linear probing. Consider this sequence:

- Insert $k_1$, where $h(k_1) = 3$, at location 3.
- Insert $k_2$, where $h(k_2) = 3$, at location 4.
- Insert $k_3$, where $h(k_3) = 3$, at location 5.
- Delete $k_2$ from location 4 by setting location 4 to empty.
- Search for $k_3$. Incorrectly stops searching at location 4 and declares $k_3$ not in the table!

Solution: when an element is deleted, instead of marking the slot as empty, it should be marked in a special way to indicate that an element used to be there but was deleted. Then the search algorithm needs to continue searching if it finds one of those slots.
Good Hash Functions for Open Addressing

An ideal hash function for open addressing would satisfy an even stronger property than that for chaining, namely:

Each key should be equally likely to have each permutation of \( \{0, 1, \ldots, M-1\} \) as its probe sequence.

This is even harder to achieve in practice than the ideal property for chaining.

A good approximation is double hashing with this scheme:

- Let \( M \) be prime, then let \( h_1(k) = k \mod M \) and let \( h_2(k) = 1 + k \mod (M - 2) \).

Generalizes the earlier example.
Average Case Analysis of Open Addressing

In this situation, the load factor $\lambda = n/M$ is always less than 1: there cannot be more keys in the table than there are table entries, since keys are stored directly in the table.

Assume that there is always at least one empty slot.

Assume that the hash function ensures that each key is equally likely to have each permutation of $\{0, 1, \ldots, M - 1\}$ as its probe sequence.

Average case running times:

- **Unsuccessful Search:** $O(\frac{1}{1-\lambda})$.
- **Insert:** Essentially same as unsuccessful search.
- **Successful Search:** $O(\frac{1}{\lambda} \cdot \ln \frac{1}{1-\lambda})$, where $\ln$ is the natural log (base $e = 2.7\ldots$).
- **Delete:** Essentially same as search.

The reasoning behind these formulas requires more sophisticated probability than for chaining.
Sanity Check for Open Addressing Analysis

The time for searches should increase as the load factor increases.

The formula for unsuccessful search is $O\left(\frac{1}{1-\lambda}\right)$.

- As $n$ gets closer to $M$, $\lambda$ gets closer to 1,
- so $1 - \lambda$ gets closer to 0,
- so $\frac{1}{1-\lambda}$ gets larger.

At the extreme, when $n = M - 1$, the formula $\frac{1}{1-\lambda} = M$, meaning that you will search the entire table before discovering that the key is not there.