

CS583 Lecture 06

Jana Kosecka

Structures for Dynamic Sets

Many slides here are based on E. Demaine , D. Luebke slides

Review: Radix Sort

- Radix sort:

Assumption: input has d digits ranging from 0 to k

Basic idea:

Sort elements by digit starting with *least* significant

Use a stable sort (like counting sort) for each stage

Each pass over n numbers with d digits takes time $O(n + k)$, so total time $O(dn + dk)$

When d is constant and $k = O(n)$, takes $O(n)$ time

Fast! Stable! Simple!

Doesn't sort in place

Review: Bucket Sort

- Bucket sort
 - Assumption: input is n reals from $[0, 1)$
 - Basic idea:
 - Create n linked lists (*buckets*) to divide interval $[0,1)$ into subintervals of size $1/n$
 - Add each input element to appropriate bucket and sort buckets with insertion sort
 - Uniform input distribution $\rightarrow O(1)$ bucket size
 - Therefore the expected total time is $O(n)$
 - These ideas will return when we study *hash tables*

Review: Order Statistics

- The i th *order statistic* in a set of n elements is the i th smallest element
- The *minimum* is thus the 1st order statistic
- The *maximum* is (duh) the n th order statistic
- The *median* is the $n/2$ order statistic
 - If n is even, there are 2 medians
- Could calculate order statistics by sorting
 - Time: $O(n \lg n)$ w/ comparison sort
 - We can do better

Review: The Selection Problem

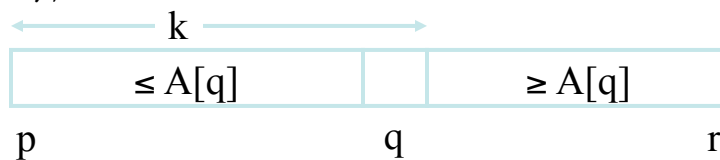
- The *selection problem*: find the i th smallest element of a set
- Two algorithms:
 - A practical randomized algorithm with $O(n)$ expected running time
 - A cool algorithm of theoretical interest only with $O(n)$ worst-case running time

Review: Randomized Selection

```

RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];    // not in
    book
    if (i < k) then
        return RandomizedSelect(A, p, q-1, i);
    else
        return RandomizedSelect(A, q+1, r, i-
k);

```



Review: Randomized Selection

- Average case

For upper bound, assume i th element always falls in larger side of partition:

$$T(n) \leq \frac{1}{n} \sum_{k=0}^{n-1} T(\max(k, n-k-1)) + \Theta(n)$$

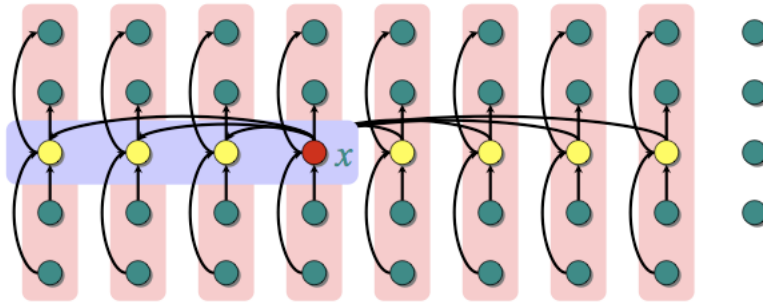
$$\leq \frac{2}{n} \sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$

We then showed that $T(n) = O(n)$ by substitution

Worst-Case Linear-Time Selection

- The algorithm in words:
 1. Divide n elements into groups of 5
 2. Find median of each group (*How? How long?*)
 3. Use Select() recursively to find median x of the $\lfloor n/5 \rfloor$ medians
 4. Partition the n elements around x . Let $k = \text{rank}(x)$
 5. **if** ($i == k$) **then** return x
 if ($i < k$) **then** use Select() recursively to find i th smallest element in first partition
 else ($i > k$) use Select() recursively to find $(i-k)$ th smallest element in last partition

Worst-Case Linear-Time Selection



At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$ group medians.

- Therefore, at least $3 \lfloor n/10 \rfloor$ elements are $\leq x$.
- Similarly, at least $3 \lfloor n/10 \rfloor$ elements are $\geq x$.

Worst-Case Linear-Time Selection

- (Sketch situation on the board)
- *How many of the 5-element medians are $\leq x$?*
At least $1/2$ of the medians $= \lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$
- *How many elements are $\leq x$?*
At least $3 \lfloor n/10 \rfloor$ elements
- For large n , $3 \lfloor n/10 \rfloor \geq n/4$ (How large?)
- So at least $n/4$ elements $\leq x$
- Similarly: at least $n/4$ elements $\geq x$

Worst-Case Linear-Time Selection

<u>$T(n)$</u>	SELECT(i, n)
$\Theta(n)$	{ 1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
$T(n/5)$	{ 2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.
$\Theta(n)$	{ 3. Partition around the pivot x . Let $k = \text{rank}(x)$.
$T(3n/4)$	{ 4. if $i = k$ then return x elseif $i < k$ then recursively SELECT the i th smallest element in the lower part else recursively SELECT the $(i-k)$ th smallest element in the upper part

Worst-Case Linear-Time Selection

- Thus after partitioning around x , step 5 will call Select() on at most $3n/4$ elements
- The recurrence is therefore:

$$\begin{aligned}
 T(n) &\leq T(\lfloor n/5 \rfloor) + T(3n/4) + \Theta(n) \\
 &\leq T(n/5) + T(3n/4) + \Theta(n) \\
 &\leq cn/5 + 3cn/4 + \Theta(n) && \lfloor n/5 \rfloor \leq n/5 \\
 &= 19cn/20 + \Theta(n) && \text{Substitute } T(n) = cn \\
 &= cn - (cn/20 - \Theta(n)) && \text{Combine fractions} \\
 &\leq cn \quad \text{if } c \text{ is big enough} && \text{Express in desired form}
 \end{aligned}$$

c has to be 20 times larger
 then constant associated Theta(n)

What we set out to prove

Worst case linear time selection

- Why not to use groups of 3 – instead of 5 ?

Linear-Time Median Selection

- Given a “black box” $O(n)$ median algorithm, what can we do?
- i th order statistic:
 - Find median x
 - Partition input around x
 - if $(i \leq (n+1)/2)$ recursively find i th element of first half
 - else find $(i - (n+1)/2)$ th element in second half
 - $T(n) = T(n/2) + O(n) = O(n)$

Can you think of an application to sorting?

Structures...

- Done with sorting and order statistics for now
- Next part of class will focus on *data structures*
- Many applications require dynamic set that supports operations Insert, Search, Delete
- E.g. compiler symbol table – keys are the identifier strings
- One options static array A – size is the number of all possible keys (very large an

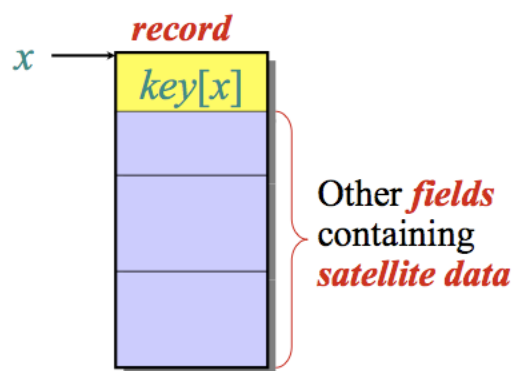
Review: Hashing Tables

- Motivation: symbol tables
- A compiler uses a *symbol table* to relate symbols to associated data
- Symbols: variable names, procedure names,
- Associated data: memory location, call graph, etc.
- For a symbol table (also called a *dictionary*), we care about search, insertion, and deletion
- We typically don't care about sorted order

Review: Hash Tables

- More formally:
- Given a table T and a record x , with key (= symbol) and satellite data, we need to support:
 - $\text{Insert}(T, x)$
 - $\text{Delete}(T, x)$
 - $\text{Search}(T, x)$
 We want these to be fast, but don't care about sorting the records
- The structure we will use is a *hash table*
Supports all the above in $O(1)$ expected time!

Review: Hash Tables



Operations on S :

- $\text{INSERT}(S, x)$
- $\text{DELETE}(S, x)$
- $\text{SEARCH}(S, k)$

Review: Hash Tables

- Example maintain 250 IP addresses of active customers of your web service
- Each IP 32-bit number 128.32.168.80
- How to organize the customers so we can retrieve , add, delete them fast
- Option 1: array indexed by IP address
- Option 2: linked list of all addresses

Hashing: Keys

- In the following discussions we will consider all keys to be (possibly large) natural numbers
- *How can we convert floats to natural numbers for hashing purposes?*
- *How can we convert ASCII strings to natural numbers for hashing purposes? (radix notation)*

Review: Direct Addressing

- Suppose
- The range of keys is $0..m-1$
- Keys are distinct
- The idea:
 Set up an array $T[0..m-1]$ in which

$$T[i] = x \quad \text{if } x \in T \text{ and key}[x] = i$$

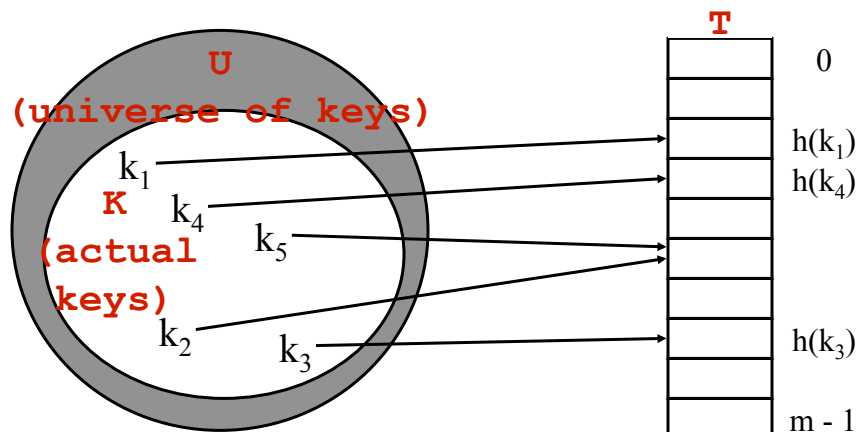
$$T[i] = \text{NULL} \quad \text{otherwise}$$
 This is called a *direct-address table*
 Operations take $O(1)$ time!
So what's the problem?

The Problem With Direct Addressing

- Direct addressing works well when the range m of keys is relatively small
- But what if the keys are 32-bit integers?
Problem 1: direct-address table will have 2^{32} entries, more than 4 billion
Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be
- Solution: map keys to smaller range $0..m-1$
- This mapping is called a *hash function*

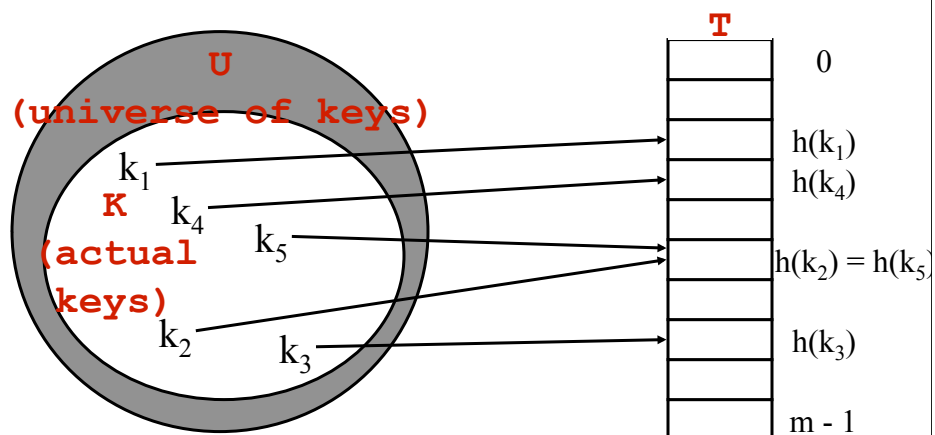
Hash Functions

- Use hash function to map U into $\{0, 1, \dots, m-1\}$



Hash Functions

- What happens when the slot is occupied – *collision*

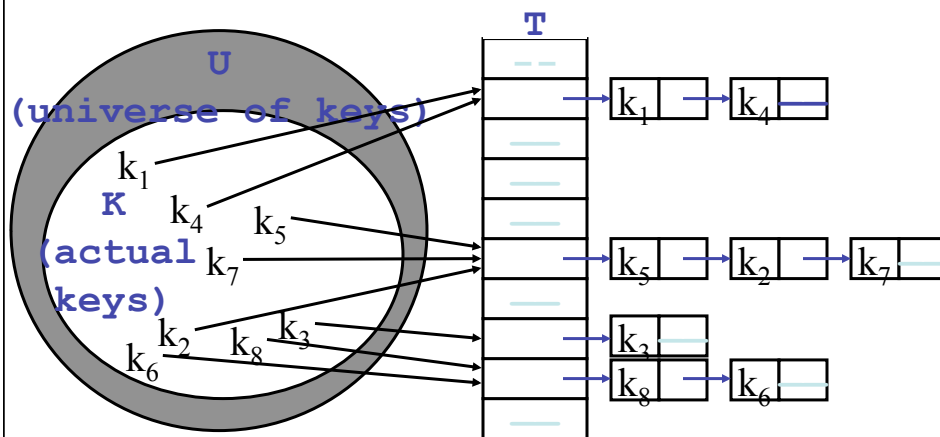


Resolving Collisions

- *How can we solve the problem of collisions?*
- Solution 1: *chaining*
- Solution 2: *open addressing*

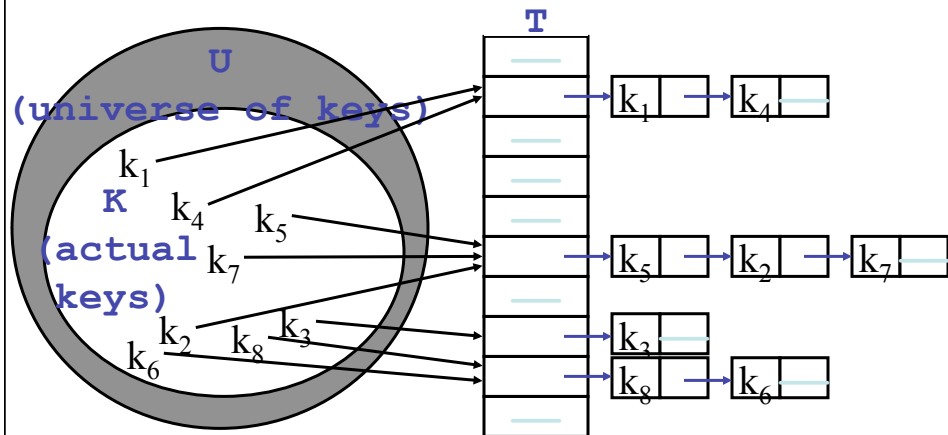
Chaining

- Chaining puts elements that hash to the same slot in a linked list:



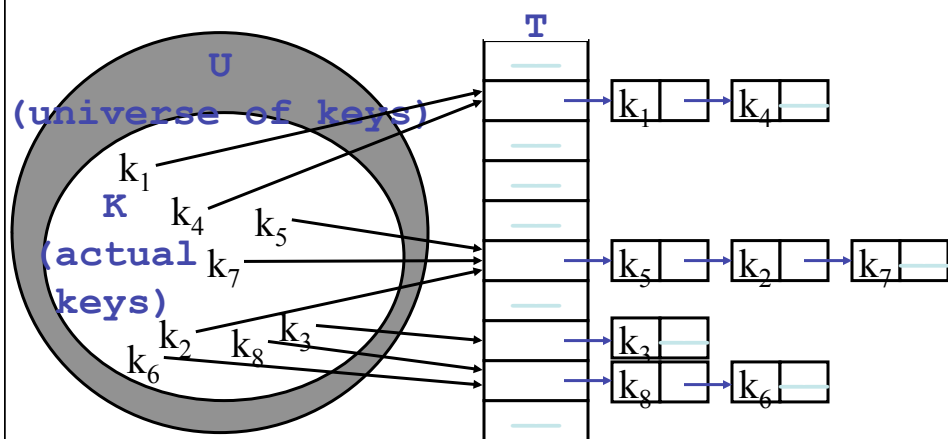
Chaining

- How do we insert an element?



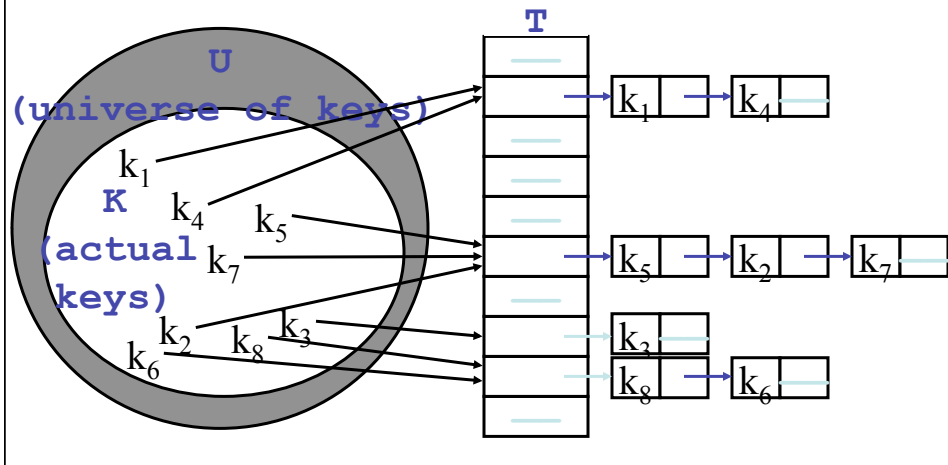
Chaining

- How do we insert an element? Worst time $O(1)$



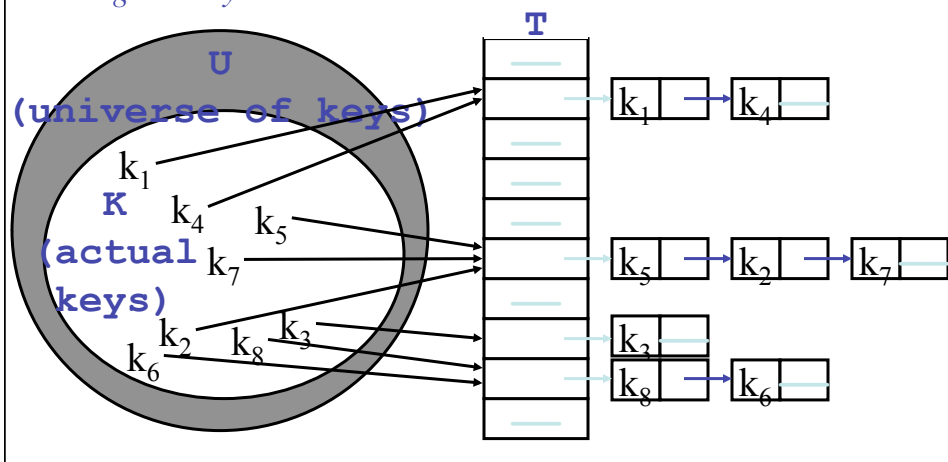
Chaining

- How do we delete an element?
Do we need a doubly-linked list for efficient delete? (yes)



Chaining

- How do we search for a element with a given key?



Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table: the *load factor* $\alpha = n/m =$ average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?*

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table, the *load factor* $\alpha = n/m =$ average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table, the *load factor* $\alpha = n/m =$ average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$
- *What will be the average cost of a successful search?*

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table, the *load factor* $\alpha = n/m =$ average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$
- Each list is equally likely be searched, α - *average length of the list*
- *What will be the average cost of a successful search?*
A: $O(1 + \alpha/2) = O(1 + \alpha)$
- Slightly different analysis list to be searched is proportional to the expected number of elements in it – expected size to searched is $\alpha/2$

Analysis of Chaining Continued

- So the cost of searching = $O(1 + \alpha)$
- *If the number of keys n is proportional to the number of slots in the table, what is α ?*
- A: $\alpha = O(1)$
In other words, we can make the expected cost of searching constant if we make α constant

Open Addressing

- Basic idea (details in Section 12.4):
- To insert: if slot is full, try another slot, ..., until an open slot is found (*probing*)
- To search, follow same sequence of probes as would be used when inserting the element
- If reach element with correct key, return it
- If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)
Example: spell checking
- Table needn't be much bigger than n
- *We will return to this later*

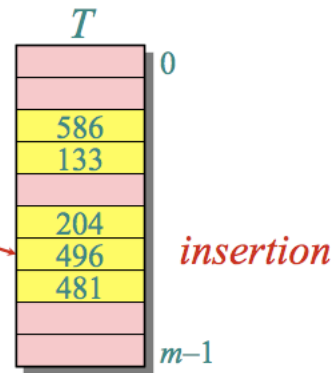
Open Addressing

Insert key $k = 496$:

0. Probe $h(496,0)$

1. Probe $h(496,1)$

2. Probe $h(496,2)$



Choosing A Hash Function

- Clearly choosing the hash function well is crucial
- *What will a worst-case hash function do?*
- *What will be the time to search in this case?*
- *What are desirable features of the hash function*
- Should distribute keys uniformly into slots
- Should not depend on patterns in the data, i.e. regularity in the data should not affect its uniformity (e.g. all even numbers)
- Three methods: hashing by division, multiplication, universal hashing

Hash Functions: The Division Method

- $h(k) = k \bmod m$
- In words: hash k into a table with m slots using the slot given by the remainder of k divided by m
- *What happens to elements with adjacent values of k ?*
- *What happens if m is a power of 2 (say 2^p)?*
- *What if m is a power of 10 ?*
- Upshot: pick table size m = prime number not too close to a power of 2 (or 10)

Hash Functions: The Division Method

- $h(k) = k \bmod m$
- In words: hash k into a table with m slots using the slot given by the remainder of k divided by m
- *What happens to elements with adjacent values of k ?*
- *What happens if m is a power of 2 (say 2^p) – hashing on p lower order bits ?*
- *What if m is a power of 10? – hashing on p least sign. Digits*
- *What if m is divisible by two and all numbers are even ?*

Hash Functions: The Division Method

- $h(k) = k \bmod m$
- Upshot: pick table size m = prime number not too close to a power of 2 (or 10), given some desirable load factor
- (e.g. 2000 elements, load factor around 3, $2000/3$
- 701 is a prime number which is close to $2000/3$, but not near any power of 2)

Hash Functions: The Multiplication Method

- For a constant A , $0 < A < 1$:
- $h(k) = \lfloor m (kA - \lfloor kA \rfloor) \rfloor$


 What does this term represent?

Hash Functions: The Multiplication Method

- For a constant A , $0 < A < 1$:
- $h(k) = \lfloor m (kA - \underbrace{\lfloor kA \rfloor}_{\text{Fractional part of } kA}) \rfloor$
- $h(k) = \lfloor m (k A \bmod 1) \rfloor$
- Value of m is not critical, Choose $m = 2^p$
- Choose A not too close to 0 or 1
- Knuth: Good choice for $A = (\sqrt{5} - 1)/2$
- Example

Hash Functions: The Multiplication Method

- For a constant A , $0 < A < 1$:
- $h(k) = \lfloor m (kA - \underbrace{\lfloor kA \rfloor}_{\text{Fractional part of } kA}) \rfloor \quad h(k) = \lfloor m (k A \bmod 1) \rfloor$
- $h(k) = (A k \bmod s) \text{ rsh}(w - p)$ where $s = 2^w$, $m = 2^p$

Open Addressing

- Basic idea (details in Section 12.4):
- To insert: if slot is full, try another slot, ..., until an open slot is found (*probing*)
- To search, follow same sequence of probes as would be used when inserting the element
- If reach element with correct key, return it
- If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)
Example: spell checking
- Table needn't be much bigger than n
- *We will return to this later*

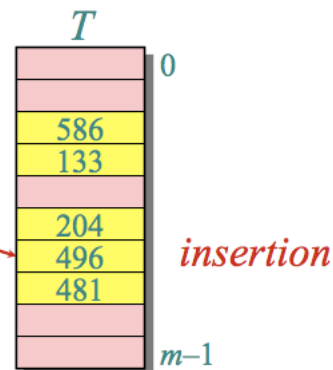
Open Addressing

Insert key $k = 496$:

0. Probe $h(496,0)$

1. Probe $h(496,1)$

2. Probe $h(496,2)$



Open Addressing

- Basic idea (details in Section 12.4):
- To insert: if slot is full, try another slot, ..., until an open slot is found (*probing*)
- Idea: for every key define a probe sequence
- $h(k,0), h(k,1), h(k,2), h(k,3) \dots$
- Linear probing

$$h(k,i) = (h'(k) + i) \bmod m$$

- Quadratic probing

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

- Double hashing

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m$$

Hash Functions: Worst Case Scenario

- Scenario:
You are given an assignment to implement hashing
You will self-grade in pairs, testing and grading your partner's implementation
In a blatant violation of the honor code, your partner:
Analyzes your hash function
Picks a sequence of “worst-case” keys, causing your implementation to take $O(n)$ time to search
- *What's an honest CS student to do?*

Review: Choosing A Hash Function

- Choosing the hash function well is crucial
 - Bad hash function puts all elements in same slot
 - A good hash function:
 - Should distribute keys uniformly into slots
 - Should not depend on patterns in the data
- We discussed three methods:
 - Division method
 - Multiplication method
 - Universal hashing

Review: The Division Method

- $h(k) = k \bmod m$
 - In words: hash k into a table with m slots using the slot given by the remainder of k divided by m
- Elements with adjacent keys hashed to different slots:
good
- If keys bear relation to m : bad
- Upshot: pick table size m = prime number not too close to a power of 2 (or 10)

Review: The Multiplication Method

- For a constant A , $0 < A < 1$:
- $h(k) = \lfloor m (kA - \lfloor kA \rfloor) \rfloor$
 $\underbrace{\hspace{1.5cm}}$
 Fractional part of kA
- Upshot:
 Choose $m = 2^P$
 Choose A not too close to 0 or 1
 Knuth: Good choice for $A = (\sqrt{5} - 1)/2$

Hash Functions: Universal Hashing

- As before, when attempting to foil an malicious adversary: randomize the algorithm
- *Universal hashing*: pick a hash function randomly in a way that is independent of the keys that are actually going to be stored
- Guarantees good performance on average, no matter what keys adversary chooses

Universal Hashing

- Let \mathcal{H} be a (finite) collection of hash functions
...that map a given universe U of keys...
...into the range $\{0, 1, \dots, m - 1\}$.
 - \mathcal{H} is said to be *universal* if:
for each pair of distinct keys $x, y \in U$,
the number of hash functions $h \in \mathcal{H}$
for which $h(x) = h(y)$ is $|\mathcal{H}|/m$
- In other words:
With a random hash function from \mathcal{H} , the chance
of a collision between x and y is exactly $1/m$
($x \neq y$)

Universal Hashing

- Theorem 11.3:
Choose h from a universal family of hash functions
Hash n keys into a table of m slots, $n \leq m$
Then the expected number of collisions involving a
particular key x is less than 1 (is less than n/m)

Proof:

For each pair of keys y, z , let $c_{yx} = 1$ if y and z collide, 0
otherwise

$E[c_{yx}] = 1/m$ (by definition)

Let C_x be total number of collisions involving key x

$$E[C_x] = \sum_{\substack{y \in U \\ y \neq x}} E[c_{yx}] = \frac{n-1}{m}$$

Since $n \leq m$, we have $E[C_x] < 1$

A Universal Hash Function

- How to design an universal class of hash functions
- Choose table size m to be prime
- Decompose key x into $r+1$ digits, so that

$$x = \{x_0, x_1, \dots, x_r\}$$

Only requirement is that max value of digit $< m$
(representation in terms of base of m)

Let $a = \{a_0, a_1, \dots, a_r\}$ denote a sequence of $r+1$
 elements chosen randomly from $\{0, 1, \dots, m-1\}$

Define corresponding hash function $h_a \in \mathcal{H}$:

$$h_a(x) = \sum_{i=0}^r a_i x_i \bmod m$$

With this definition, \mathcal{H} has m^{r+1} members

A Universal Hash Function

- \mathcal{H} is a universal collection of hash functions
 (Theorem 12.4)
- How to use:
 - Pick r based on m and the range of keys in U
 - Pick a hash function by (randomly) picking the a 's
 - Use that hash function on all keys

The end

Dynamic Sets

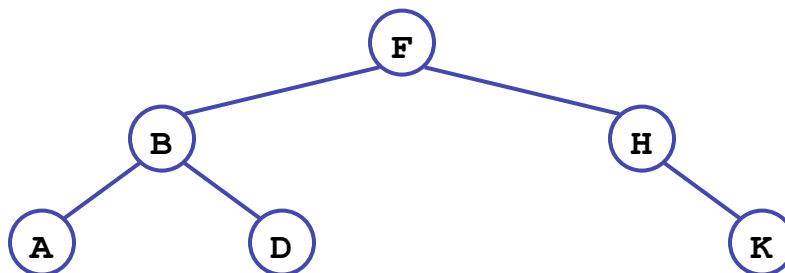
- Another example of data structure
- In particular, structures for *dynamic sets*
 - Elements have a *key* and *satellite data*
 - Dynamic sets support *queries* such as:
 - Search***(S, k), ***Minimum***(S), ***Maximum***(S),
 - Successor***(S, x), ***Predecessor***(S, x)
 - They may also support *modifying operations* like:
 - Insert***(S, x), ***Delete***(S, x)

Binary Search Trees

- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, elements have:
 - key*: an identifying field inducing a total ordering
 - left*: pointer to a left child (may be NULL)
 - right*: pointer to a right child (may be NULL)
 - p*: pointer to a parent node (NULL for root)

Binary Search Trees

- BST property:
 $\text{key}[\text{left}(x)] \leq \text{key}[x] \leq \text{key}[\text{right}(x)]$
- Example:



Inorder Tree Walk

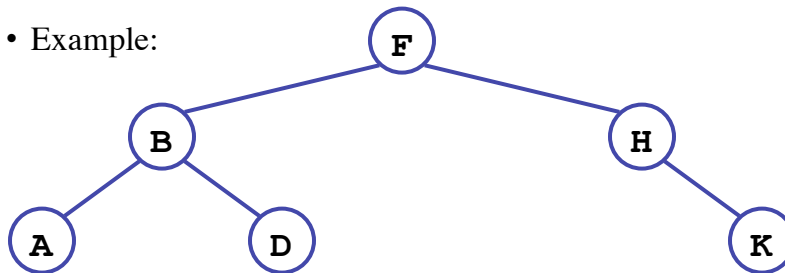
- *What does the following code do?*

```
TreeWalk(x)
    TreeWalk(left[x]);
    print(x);
    TreeWalk(right[x]);
```

- A: prints elements in sorted (increasing) order
- This is called an *inorder tree walk*
Preorder tree walk: print root, then left, then right
Postorder tree walk: print left, then right, then root

Inorder Tree Walk

- Example:



- *How long will a tree walk take?*
- *Prove that inorder walk prints in monotonically increasing order*

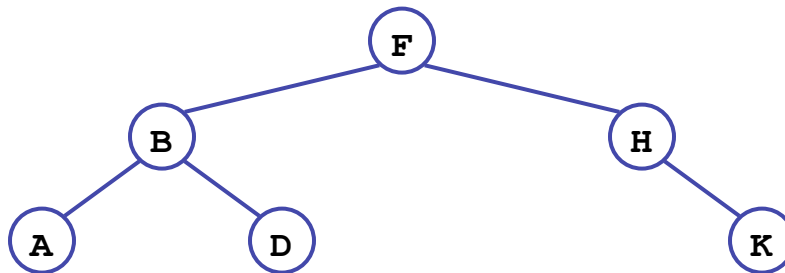
Operations on BSTs: Search

- Given a key and a pointer to a node, returns an element with that key or NULL:

```
TreeSearch(x, k)
    if (x = NULL or k = key[x])
        return x;
    if (k < key[x])
        return TreeSearch(left[x],
k);
    else
        return TreeSearch(right[x],
k);
```

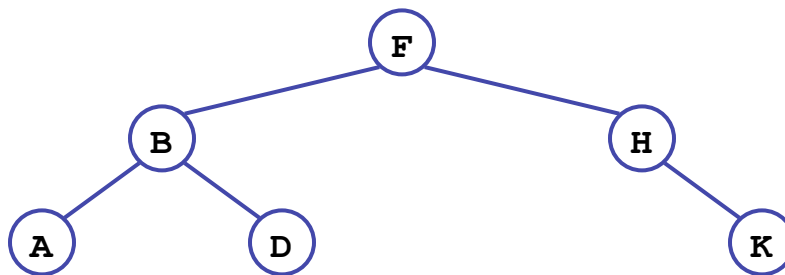
BST Search: Example

- Search for *D* and *C*:



BST Search: Example

- Search for *D* and *C*:

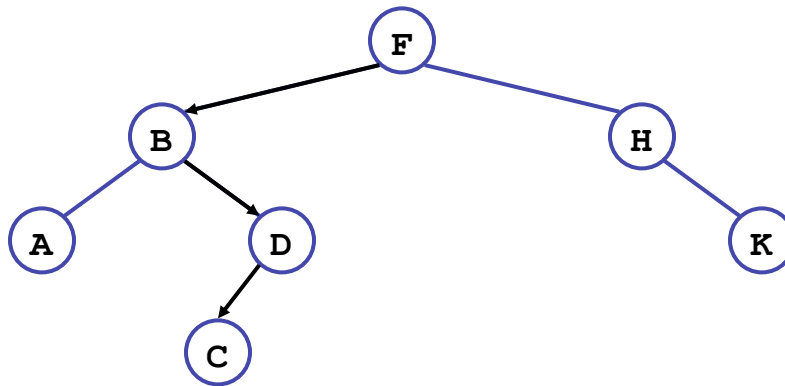


Operations of BSTs: Insert

- Adds an element *x* to the tree so that the binary search tree property continues to hold
- The basic algorithm
- Like the search procedure above
- Insert *x* in place of NULL
Use a “trailing pointer” to keep track of where you came from (like inserting into singly linked list)

BST Insert: Example

- Example: Insert C



BST Search/Insert: Running Time

- *What is the running time of `TreeSearch()` or `TreeInsert()`?*
- A: $O(h)$, where h = height of tree
- *What is the height of a binary search tree?*
- A: worst case: $h = O(n)$ when tree is just a linear string of left or right children
We'll keep all analysis in terms of h for now
Later we'll see how to maintain $h = O(\lg n)$

Sorting With Binary Search Trees

- Informal code for sorting array A of length n :

```

BSTSort(A)
  for  $i=1$  to  $n$ 
    TreeInsert(A[i]);
  InorderTreeWalk(root);
  
```

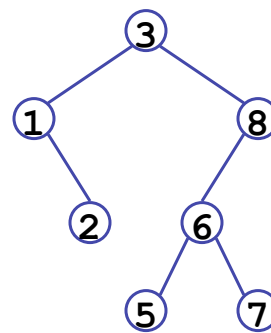
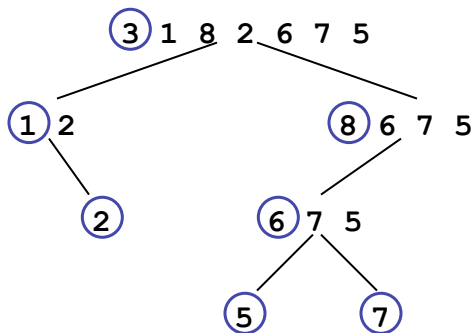
- Argue that this is $\Omega(n \lg n)$
- What will be the running time in the Worst case?
- Average case? (hint: remind you of anything?)

Sorting With BSTs

- Average case analysis
It's a form of quicksort!

```

for  $i=1$  to  $n$ 
  TreeInsert(A[i]);
InorderTreeWalk(root);
  
```



Sorting with BSTs

- Same partitions are done as with quicksort, but in a different order
 - In previous example
 - Everything was compared to 3 once
 - Then those items < 3 were compared to 1 once
 - Etc.
 - Same comparisons as quicksort, different order!
 - Example: consider inserting 5

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: $O(n \lg n)$
- *Which do you think is better, quicksort or BSTsort? Why?*

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: $O(n \lg n)$
- *Which do you think is better, quicksort or BSTSort? Why?*
- A: quicksort
 - Better constants
 - Sorts in place
 - Doesn't need to build data structure

More BST Operations

- BSTs are good for more than sorting. For example, can implement a priority queue
- *What operations must a priority queue have?*
 - Insert
 - Minimum
 - Extract-Min

BST Operations: Minimum

- *How can we implement a Minimum() query?*
- *What is the running time?*

BST Operations: Successor

- For deletion, we will need a Successor() operation
- Draw Fig 13.2
- *What is the successor of node 3? Node 15? Node 13?*
- *What are the general rules for finding the successor of node x? (hint: two cases)*

BST Operations: Successor

- Two cases:
 - x has a right subtree: successor is minimum node in right subtree
 - x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
- Intuition: As long as you move to the left up the tree, you're visiting smaller nodes.
- Predecessor: similar algorithm

BST Operations: Delete

- Deletion is a bit tricky

- 3 cases:

x has no children:

Remove x

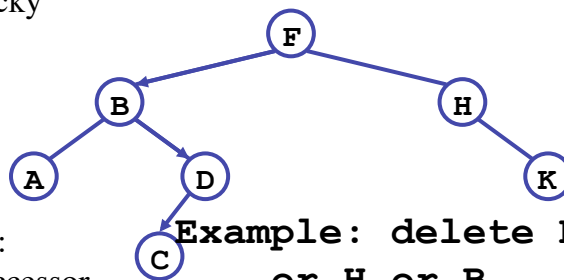
x has one child:

Splice out x

x has two children:

Swap x with successor

Perform case 1 or 2 to delete it



BST Operations: Delete

- *Why will case 2 always go to case 0 or case 1?*
- A: because when x has 2 children, its successor is the minimum in its right subtree
- *Could we swap x with predecessor instead of successor?*
- A: yes. *Would it be a good idea?*
- A: might be good to alternate

The End

- Up next: guaranteeing a $O(\lg n)$ height tree