Chapter Nine

Searching and Hashing

Revision - Searching

- Searching is a process of ***checking*** and ***finding an element*** from a list of elements.
- Let A be a collection of data elements, *i.e.,* A is a linear array of say *n* elements. If we want to find the presence of an element ***"data"*** in A, then we have to search for it.
- The search is successful if ***data*** does appear in A and unsuccessful if otherwise.
- There are several types of searching techniques; one has some advantage(s) over other.
- Following are the ***four important*** searching techniques :
  - Linear or Sequential Searching
  - Binary Searching
  - ***Interpolation Searching***
  - ***Fibonacci Searching***

Interpolation Search

- This method is even ***more efficient*** than binary search, if the elements are ***uniformly distributed*** (or sorted) in an array A.
- Its average case complexity is **O(log log N)** where N is the number of keys.
- Consider an array A of n elements and the elements are uniformly distributed (sorted).
- Initially, as in binary search, ***low is set to 0*** and ***high is set to n-1.***
- Now we are searching an element key in an array between ***A[low] and A[high].***
- The key would be expected to be at ***mid***, which is an approximately position.
  - » **mid = low + (high – low) × ((key – A[low])/(A[high] – A[low]))**

Cont'd

- If key is lower than A[mid], reset high to mid–1; else reset low to mid+1.
- Repeat the process until the key has found or low > high.
- Example: 2, 25, 35, 39, 40, 47, 50
  - » **CASE 1:** Say we are ***searching 50*** from the array.
    - Here n = 7
    - Key = 50
    - low = 0
    - high = n – 1 = 6
    - mid = 0+(6–0) × ((50–2)/(50–2))
    - = 6 × (48/48) = **6**
    - if (key == A[mid])
    - ⇒ **key == A[6]**
    - **⇒ 50 == 50**
  - » ⇒ key is found.

Cont'd

- **CASE 2**: Say we are searching 34 from the array
  - Here $n = 7$ Key = 34
  - low = 0
  - high = $n - 1 = 6$
  - mid = $0 + (6 - 0) \times ((34 - 2)/(34 - 2))$
  - $= 6 \times (32/48)$
  - = 4
  - if(key < A[mid])
  - $\Rightarrow$ key < A[4]
  - $\Rightarrow 34 < 40$
  - so reset high = mid–1
  - $\Rightarrow 3$
  - low = 0
  - high = 3

Cont'd

- Since(low < high)
- mid = 0+(3–0) × ((34–2)/(39–2))
- = 3 × (32/37)
- = **2.59**   Here we consider only the integer part of the mid
  - *i.e.*, mid = 2
- if (key < A[mid])
- ⇒ key < A[2]
- ⇒ 34 < 35
- so reset high = mid–1
- ⇒ 1
- low = 0
- high = 1

Cont'd

- Since (low < high)
- mid = 0+(1–0) $\times$ ((34–2)/(25–2))
- = 3 $\times$ (32/23)
- = 1
- here (key > A[mid])
- $\Rightarrow$ key > A[1]
- $\Rightarrow$ 34 > 25
- so reset low = mid+1
- $\Rightarrow$ 2
- low = 2
- high = 1
- Since (low > high)
- DISPLAY " The key is not in the array"
- STOP

Algorithm

- 1. Input a sorted array of *n* elements and the key to be searched
- 2. Initialize low = 0 and high = n – 1
- 3. Repeat the steps 4 through 7 until if(low < high)
- 4. Mid = low + (high – low) × ((key – A[low]) / (A[high] – A[low]))
- 5. If(key < A[mid])
  - (*a*) high = mid–1
- 6. Elseif (key > A[mid])
  - (*a*) low = mid + 1
- 7. Else
  - (*a*) DISPLAY " The key is not in the array"
  - (*b*) STOP
- 8. STOP

Implementation

```
class interpolation
{
int Key;
int Low,High,Mid;
public:
void InterSearch(int*,int);
};
//This function will search the element using interpolation search
void interpolation::InterSearch(int *Arr,int No)
{
int Key;
//Assigning the pointer low and high
Low=0;High=No-1;
//Inputting the element to be searched
cout<<"\n\nEnter the Number to be searched = ";
cin>>Key;
```

Cont'd

```
while(Low < High)
{
//Finding the Mid position of the array to be searched
Mid=Low+(High-Low)*((Key-Arr[Low])/(Arr[High]-Arr[Low]));
if (Key < Arr[Mid])
//Re-initializing the high pointer if the
//key is greater than the mid value
High=Mid-1;
else if (Key > Arr[Mid])
//Re initializing the low pointer if the
//key is less than the mid value
Low=Mid+1;
else
{
//if the key value is equal to the mid value
```

Cont’d

```
//of the array, the key is found
cout<<“\nThe key “<<Key<<” is found at the location ”<<Mid;
return;
}
};
cout<<“\n\nThe Key “<<Key<<” is NOT found”;
}
void main()
{
int *a,n,*b;
interpolation Ob;
clrscr();
cout<<“\n\nEnter the number of elements : ”;
cin>>n;
a=new int[n];
b=a;
```

Cont'd

```
//Input the elements in the array
for (int i=0;i<n;i++)
{
cout<<"\nEnter the "<<i<<" element : ";
cin>>*a;
a++;
}
//calling the InterSearch function using objects
Ob.InterSearch(b,n);
cout<<"\n\nPress any key to continue...";
getch();
}
```

Fibonacci Search

- Fibonacci Search – Reading Assignment

Hashing

- ***Hashing*** is a technique where we can compute the location of the ***desired record*** in order to retrieve it ***in a single access*** (or comparison).
- Suppose we were to come up with a ***"magic function"*** that, given ***a value to search for***, would tell us ***exactly*** where in the array to look.
  - If it's in that location, it's in the array.
  - If it's not in that location, it's not in the array.
- This function would have no other purpose.
- If we look at the function's inputs and outputs, they probably won't "make sense".
- This function is called a ***hash function*** because it ***"makes hash"*** of its inputs.

Cont'd

- Key-value pairs are stored in a fixed size table called a *hash table (symbol table)*.
  - A hash table is partitioned into many ***buckets***.
  - Each bucket has many ***slots***.
  - Each slot holds ***one record***.
  - A hash function h(x) transforms the identifier (key) into an address in the hash table
- The process of implementing this hash table (symbol table) is called ***hashing*** which is both conceptually ***simple and very efficient.***
- Search tree methods: key comparisons
  - Time complexity: O(size) or O(log n)
- Hashing methods: hash functions
  - ***Expected time: O(1)***

Cont'd

- Following are the most popular **Distribution - Independent hash functions :**
  - » Division method
    - – $H(k) = k \pmod{m}$
  - » Mid Square method
    - – $H(k) = k^2$ and the middle digits will be selected as a key.
  - » Folding method.
    - – $H(k) = k1 + k2 + \ldots\ldots + kr$

Cont'd

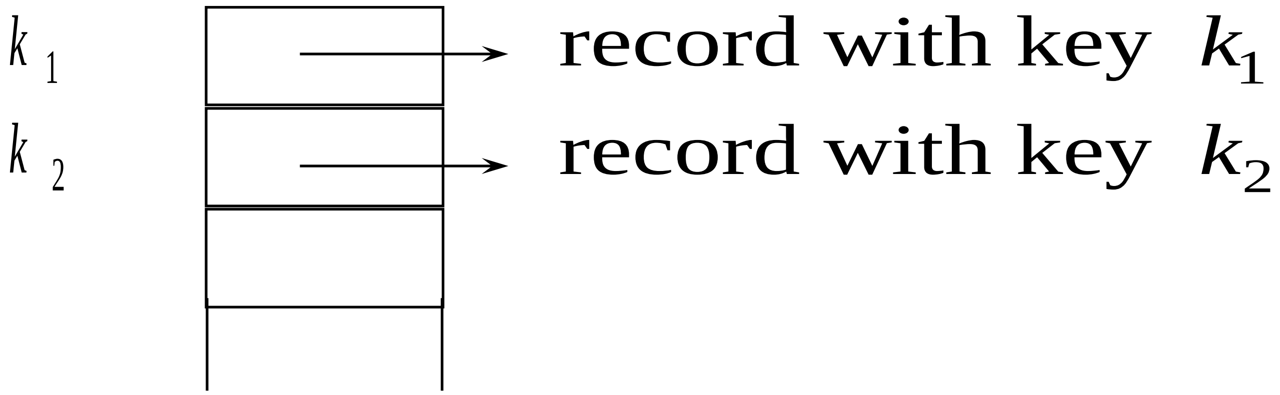
s slots

b buckets

	0	1		s-1
0			. . .	
1				
	⋮	⋮		⋮
b-1			. . .	

Cont'd

- **Observation:** We can store a set very easily if we can use its keys as array indices:
- A:



- e.g. SEARCH(A,k)
- return A[k]

Cont'd

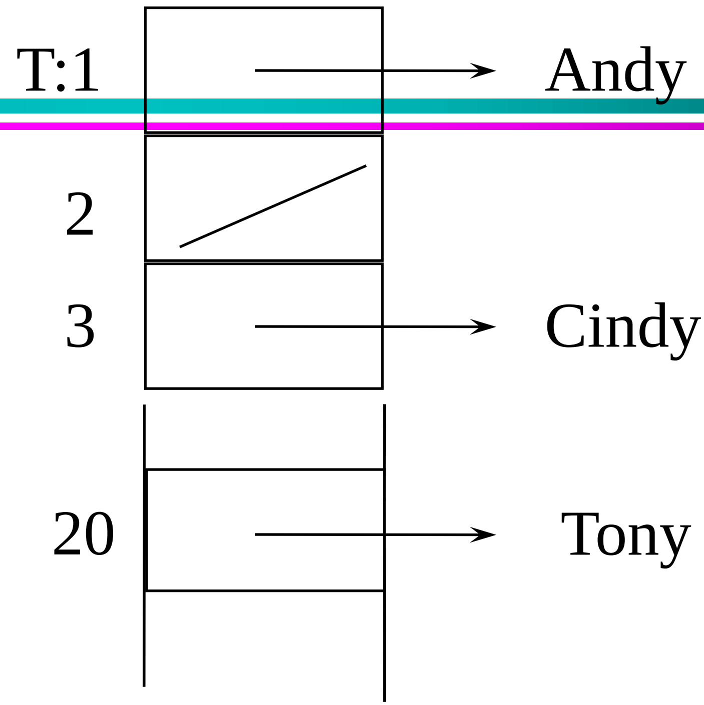
- **Problem:** usually, the number of **possible keys** is ***far larger*** than the ***number of keys actually stored***, or even than available memory. (E.g., strings.)

- **Idea of hashing**: use a **function *h*** to map keys into a smaller set of indices, say the integers 0…m. This function is called a ***hash function.***

- E.g. $h(k)$ = position of k's **first letter** in the alphabet.

$$h(\text{"}Andy\text{"}) = 1$$

$$h(\text{"}Cindy\text{"}) = 3$$



$$h(\text{"}Tony\text{"}) = 20$$

$$h(\text{"}Thomas\text{"}) = 20 \quad \ldots \text{ oops } \ldots$$

Problem: ***Collisions.*** They are ***inevitable*** if there are more possible key values than table slots.

Cont'd

- Uses an array table[0:b-1].
  - Each position of this array is a bucket.
  - A bucket can normally hold only one dictionary pair.

- Uses a hash function h that converts each key k into an index in the range [0, b-1].

- Every dictionary pair (key, element) is stored in its home bucket table[h[key]].

Cont'd

- Pairs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is table[0:7], b = 8.
- Hash function is key (mod 11).

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Cont'd

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Where does (26,g) go?
- Keys that have the same home bucket are synonyms.
  - 22 and 26 are synonyms with respect to the hash function that is in use.
- The bucket for (26,g) is already occupied.

Collisions

- When ***two values*** hash to the ***same array location***, this is called a **collision**.

- ***Collisions*** are normally treated as “first come, first served” - the ***first value*** that hashes to the location gets it.

- We have to find something to do with the second and subsequent values that hash to this same location.

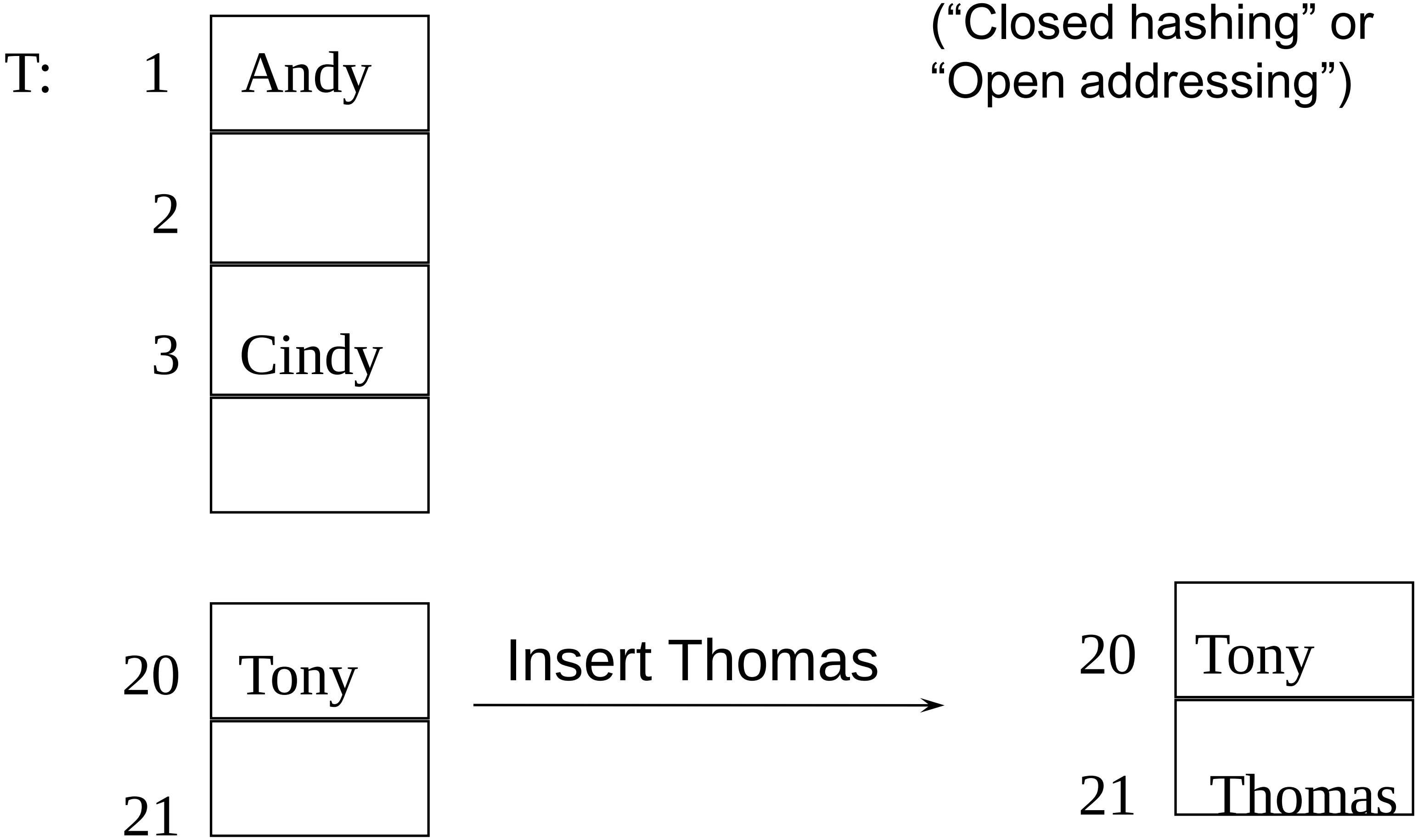
Resolving Collisions

- Let's assume for now that our hash function is OK, and deal with the collision resolution problem.
- Two groups of solutions:
  - Store the colliding key in the hash-table array. ("***Closed hashing***")
  - Store it somewhere else. ("***Open hashing***")

Closed Hashing

- Store colliders in the hash table array itself:

("Closed hashing" or
"Open addressing")



Example: Insertion I

- Suppose you want to add seagull to this hash table
- Also suppose:
  - » hashCode(seagull) = 143
  - » table[143] is not empty
  - » table[143] != seagull
  - » table[144] is not empty
  - » table[144] != seagull
  - » table[145] is empty
- Therefore, put seagull at location 145

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Searching I

- Suppose you want to look up seagull in this hash table
- Also suppose:
  - » hashCode(seagull) = 143
  - » table[143] is not empty
  - » table[143] != seagull
  - » table[144] is not empty
  - » table[144] != seagull
  - » table[145] is not empty
  - » table[145] == seagull !
- We found seagull at location 145

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Searching II

- Suppose you want to look up cow in this hash table
- Also suppose:
  - » hashCode(cow) = 144
  - » table[144] is not empty
  - » table[144] != cow
  - » table[145] is not empty
  - » table[145] != cow
  - » table[146] is empty
- If cow were in the table, we should have found it by now
- Therefore, it isn’t here

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Insertion II

- Suppose you want to add hawk to this hash table
- Also suppose
  - » hashCode(hawk) = 143
  - » table[143] is not empty
  - » table[143] != hawk
  - » table[144] is not empty
  - » table[144] == hawk
- hawk is already in the table, so do nothing

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

Insertion III

- Suppose:
  - » You want to add cardinal to this hash table
  - » hashCode(cardinal) = 147
  - » The last location is 148
  - » 147 and 148 are occupied
- Solution:
  - » Treat the table as circular; after 148 comes 0
  - » Hence, cardinal goes in location 0 (or 1, or 2, or ...)

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl

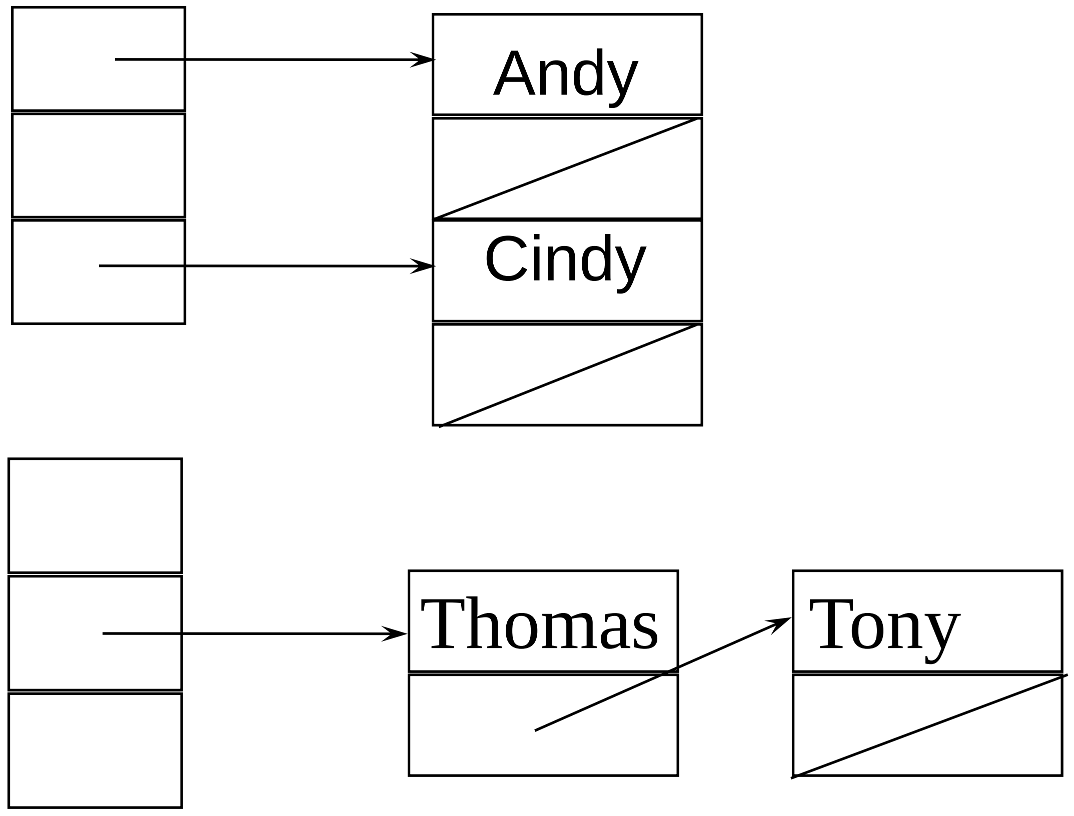
Closed Hashing…

- Advantage:
  - No extra storage for lists
- Disadvantages:
  - Harder to program
  - Harder to analyze
  - Table can overflow
  - Performance is worse

Open Hashing

- Put all the keys that hash to the same index onto a linked list. Each T[i] called a ***bucket*** or ***slot***.

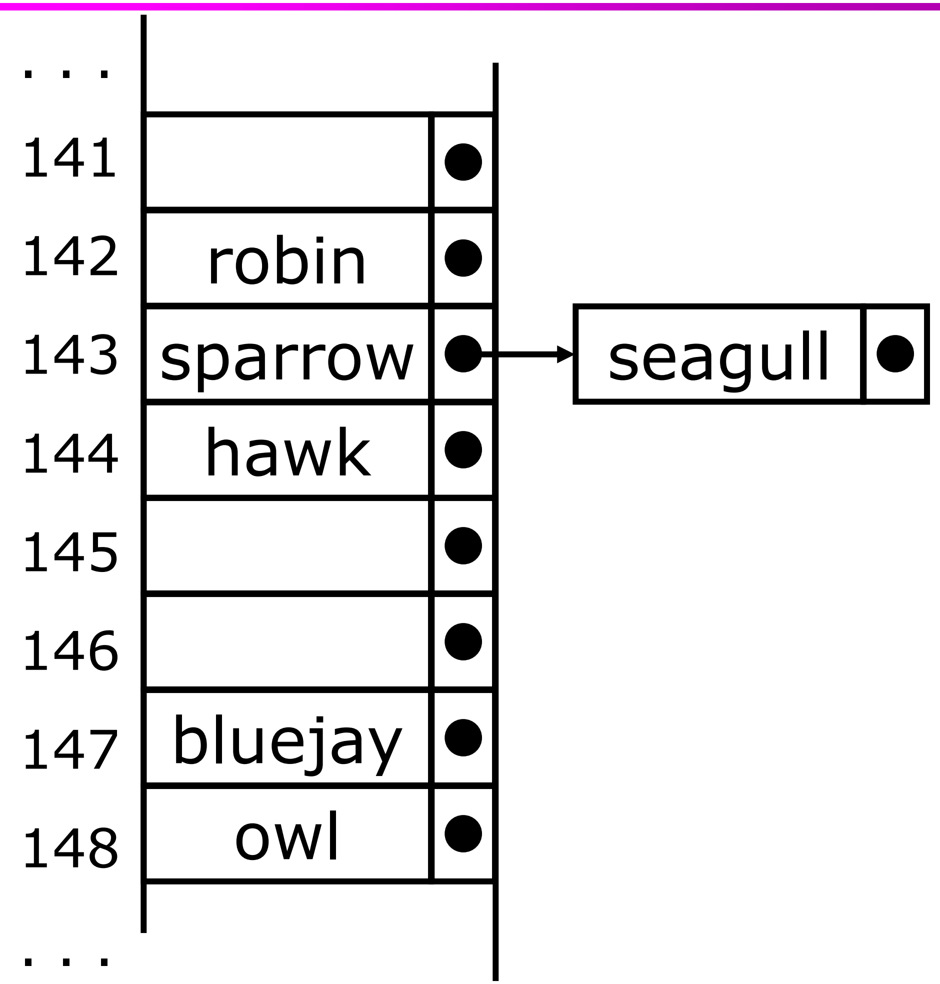
- T: 1
  2
  3



20

Example: Insertion

- The previous solutions used closed hashing: all entries went into a "flat" (unstructured) array.
- Another solution is to make each array location the header of a ***linked list*** of values that hash to that location.



Application of Hashing

- Hashing is *vastly* more prevalent than trees for in-memory storage.

- Examples:
  - UNIX shell command cache
  - “arrays” in Icon, Awk, Tcl, Perl, etc.
  - Compiler symbol tables
  - Filenames on CD-ROM
  - And more…