

# Searching

CPE 212 -- Lecture 17

# Linear Search

- Consider a sequential list of elements
- If we wish to insert elements as quickly as possible, then *InsertItem* is  $O(1)$ 
  - New element placed in next array slot
  - Element added at first position in linked list
- To find a particular value, one may have to scan the entire sequence of values  $O(N)$
- Average  $N/2$  comparisons for successful search

# Self-Organizing / Self-Adjusting Lists

- Put frequently accessed items near front
  - Strategy #1
    - Move each item accessed to the front
  - Strategy #2
    - Upon access to an element, swap it with the element that precedes it
- Worst case performance still  $O(N)$
- Average performance on successful searches improves

# Binary Search

- Assume elements are stored in a sorted array

$$A[0] < A[1] < A[2] < \dots A[n-1]$$

- Use divide and conquer to determine if a value is in the list
- $O(\log_2 n)$  search

# Binary Search Example - 1

List	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	1	3	5	7	8	10	12	15	18

**Is 15 in the array?**

# Binary Search Example - 2

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
1	3	5	7	8	10	12	15	18

first ↑ last

$$\text{midpoint} = (\text{first} + \text{last}) / 2 = 4$$

List[4] == 15    No

List[4] < 15    Yes

$$\text{first} = \text{midpoint} + 1 = 5$$

# Binary Search Example - 3

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
1	3	5	7	8	10	12	15	18



midpoint = (first+last)/2 = 6 (integer division)

List[6] == 15      No

List[6] < 15      Yes

first = midpoint + 1 = 7

# Binary Search Example - 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
1	3	5	7	8	10	12	15	18



↑  
 $\text{midpoint} = (\text{first} + \text{last}) / 2 = 7$  (integer division)

List[7] == 15    Yes

Done



# Binary Search Algorithm

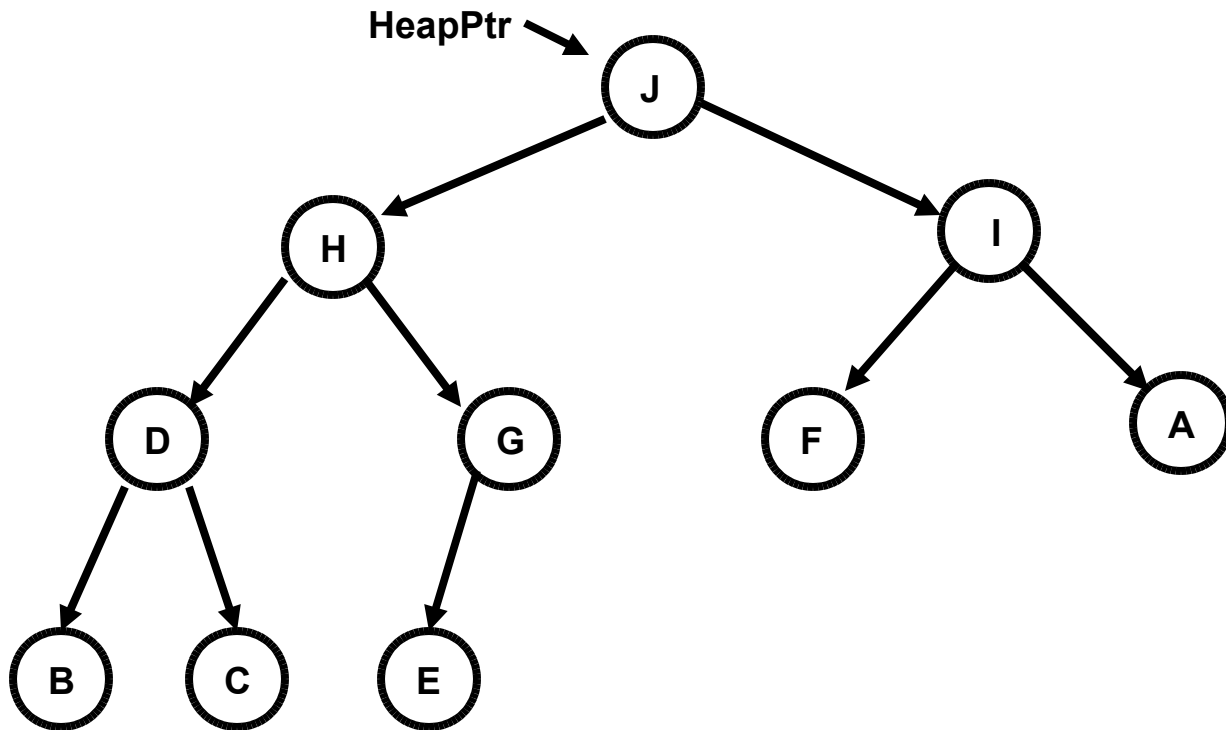
```
void SortedType::RetrieveItem(ItemType& item, bool& found)
{
    int midPoint;
    int first = 0;
    int last = length - 1;

    bool moreToSearch = first <= last;
    found = false;
    while (moreToSearch && !found)
    {
        midPoint = (first + last) / 2;
        switch (item.ComparedTo(info[midPoint]))
        {
            case LESS      : last = midPoint - 1;
                           moreToSearch = first <= last;
                           break;
            case GREATER   : first = midPoint + 1;
                           moreToSearch = first <= last;
                           break;
            case EQUAL     : found = true;
                           item = info[midPoint];
                           break;
        }
    }
}
```

# Heap ADT

- Heap
  - A complete binary tree, each of whose elements contains a value that is greater than or equal to the value of each of its children (aka maximum heap)
  - Shape property & order property
- Complete Binary Tree
  - A binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible

# Heap Example



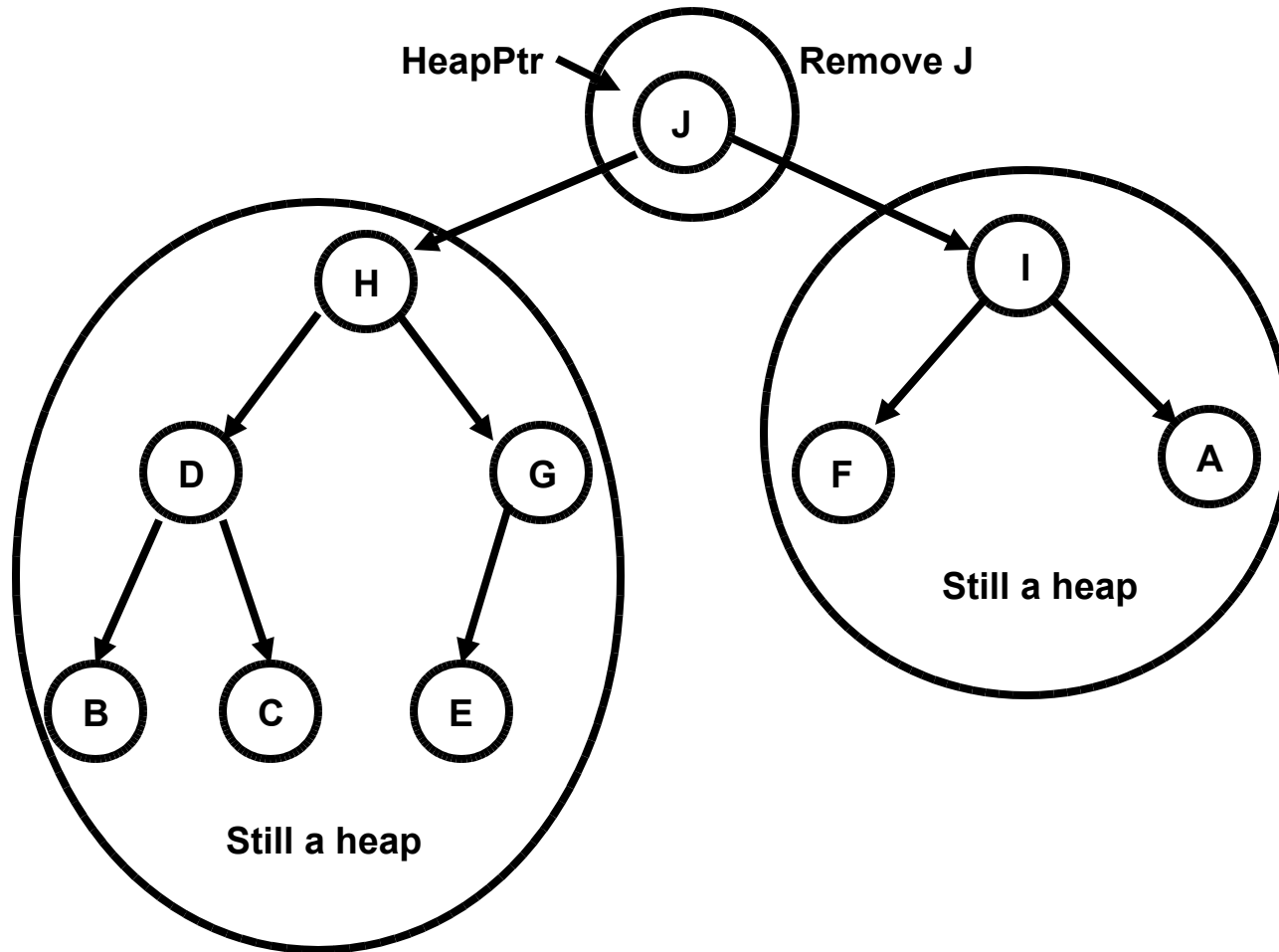
Largest Element of Heap resides in the root node

myHeap.elements

[0]	J
[1]	H
[2]	I
[3]	D
[4]	G
[5]	F
[6]	A
[7]	B
[8]	C
[9]	E

UAHuntsville

# Heap Example

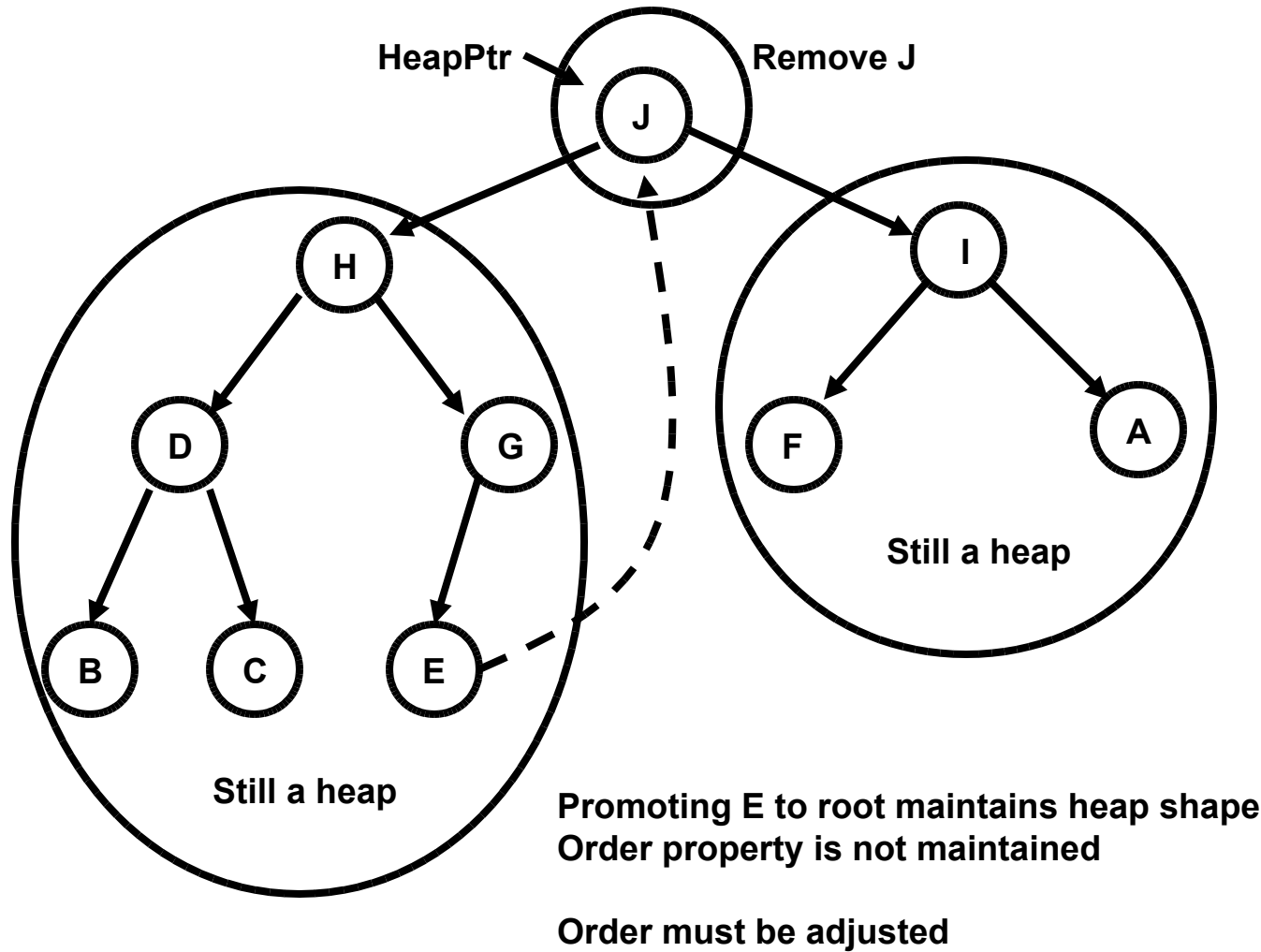


myHeap.elements

[0]	J
[1]	H
[2]	I
[3]	D
[4]	G
[5]	F
[6]	A
[7]	B
[8]	C
[9]	E

UAHuntsville

# Heap Example

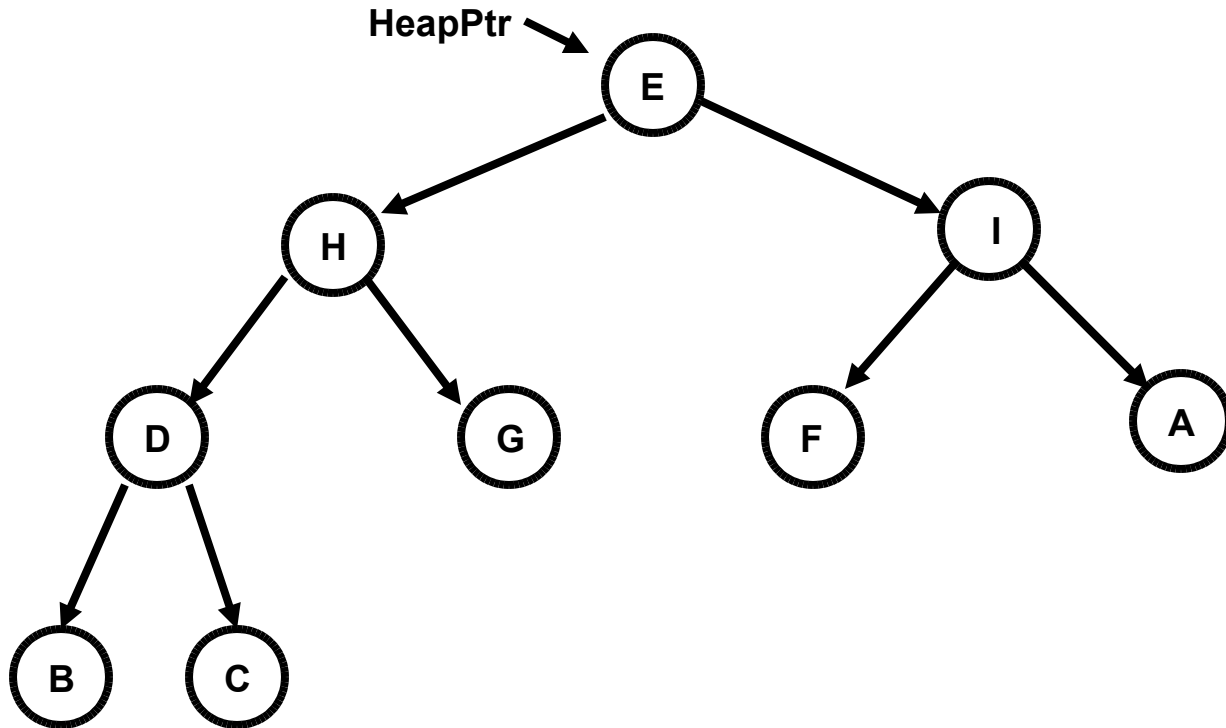


myHeap.elements

[0]	E
[1]	H
[2]	I
[3]	D
[4]	G
[5]	F
[6]	A
[7]	B
[8]	C
[9]	

UAHuntsville

# Heap Example



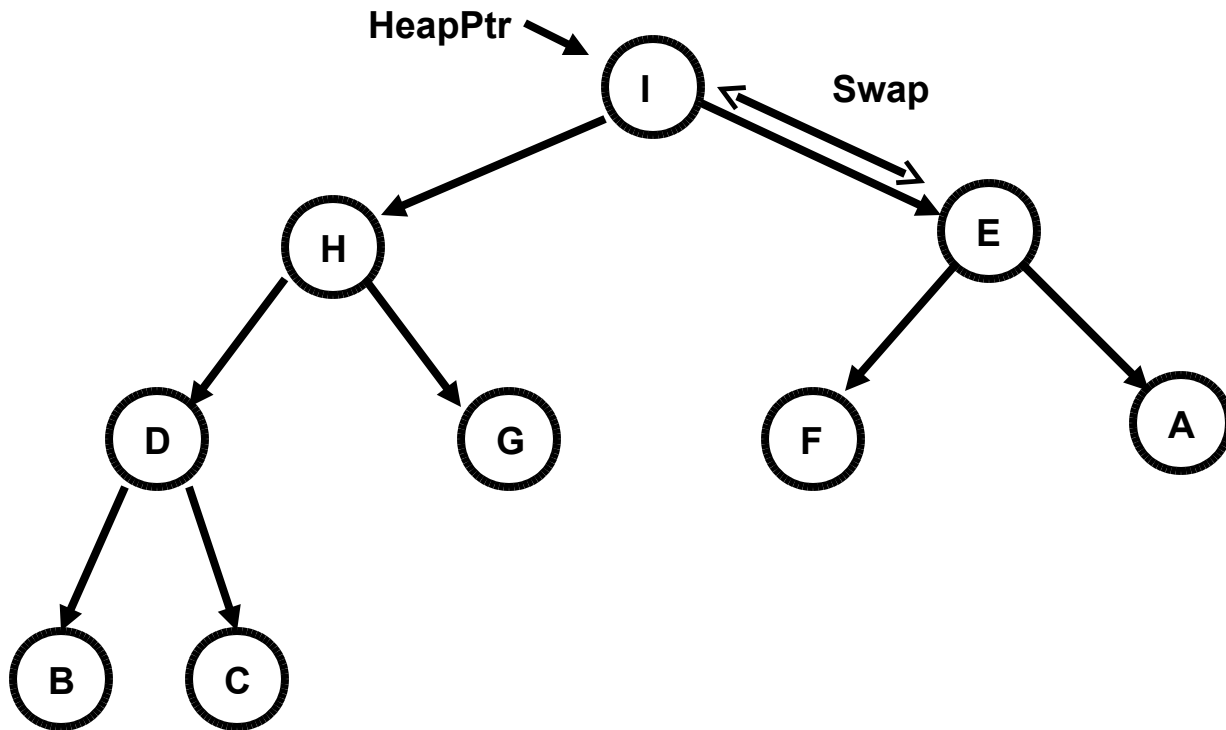
**ReheapDown** operation

myHeap.elements

[0]	E
[1]	H
[2]	I
[3]	D
[4]	G
[5]	F
[6]	A
[7]	B
[8]	C
[9]	

UAHuntsville

# Heap Example



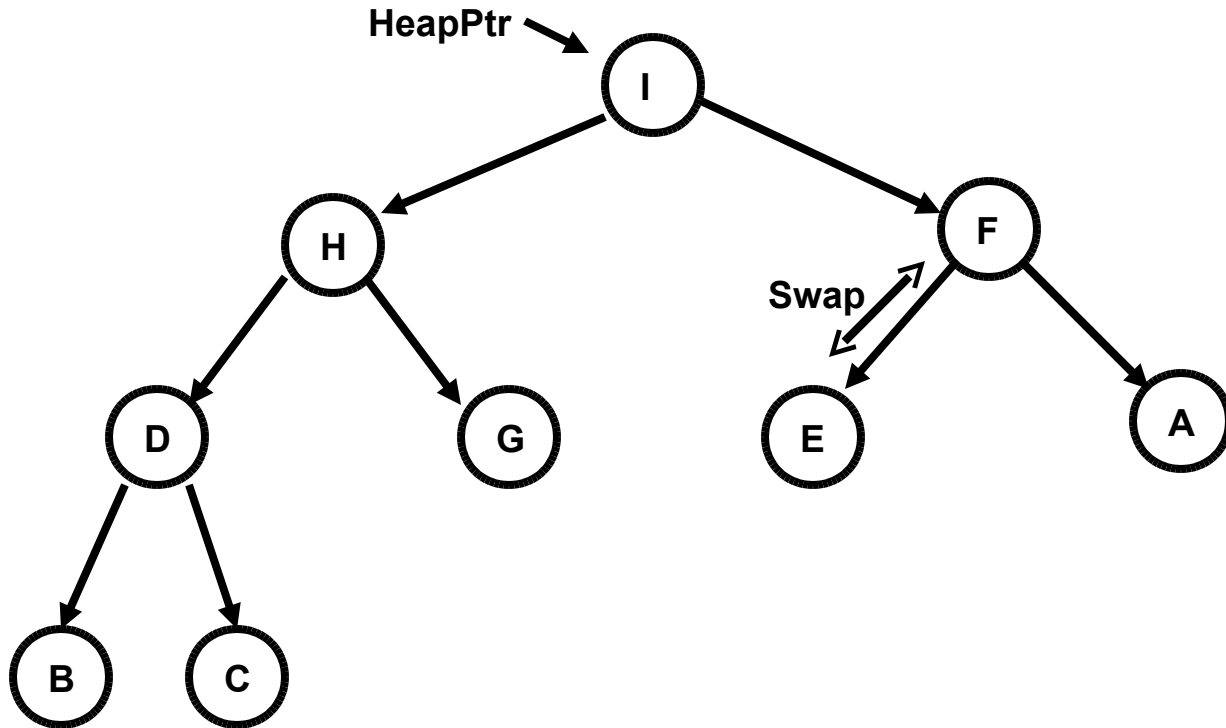
ReheapDown operation

myHeap.elements

[0]	I
[1]	H
[2]	E
[3]	D
[4]	G
[5]	F
[6]	A
[7]	B
[8]	C
[9]	

UAHuntsville

# Heap Example



**ReheapDown** operation

myHeap.elements

[0]	I
[1]	H
[2]	F
[3]	D
[4]	G
[5]	E
[6]	A
[7]	B
[8]	C
[9]	

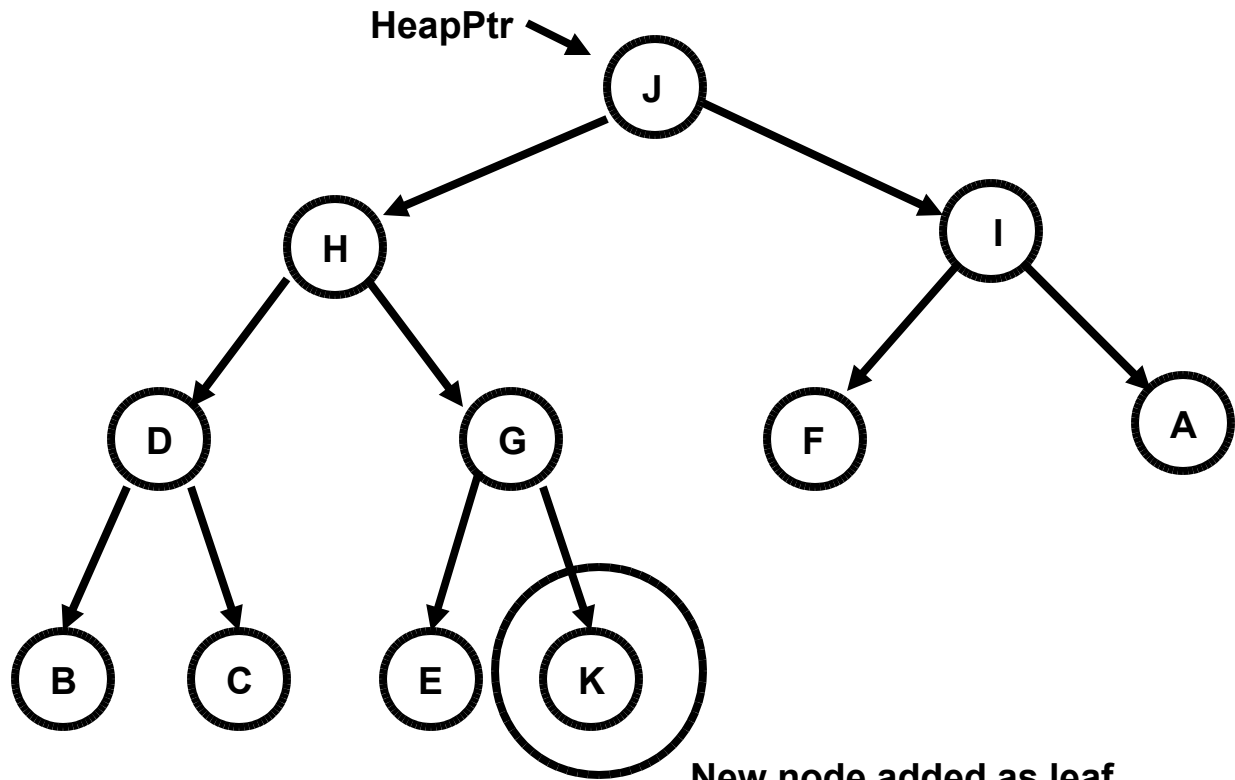
UAHuntsville



# Heap ADT

- ReheapDown(root,bottom)
  - Restores the order property of the heaps to the tree between root and bottom
  - Precondition
    - The order property of heaps may be violated only by the root node of the tree
  - Postcondition
    - The order property applies to all elements of the heap

# Heap Example



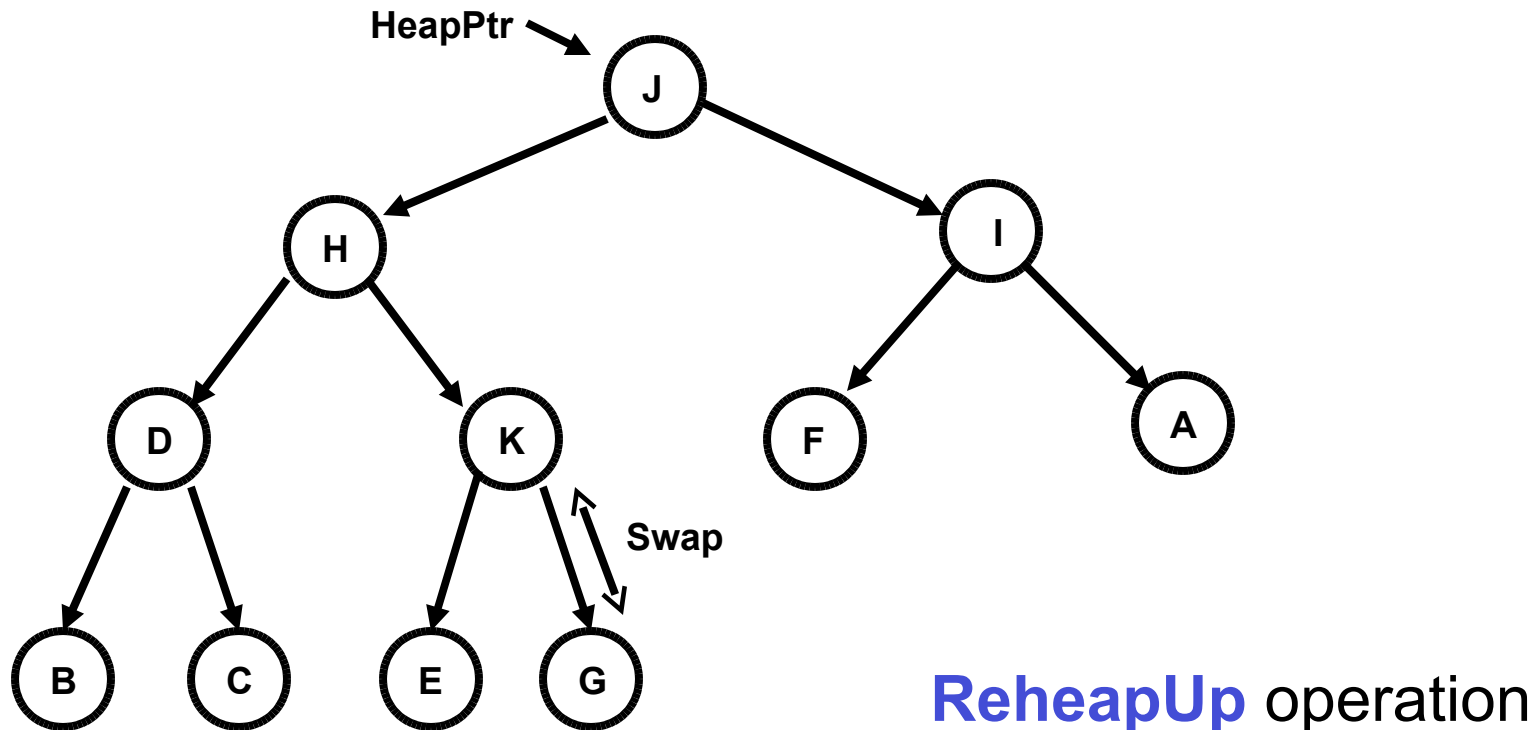
New node added as leaf

Shape property maintained

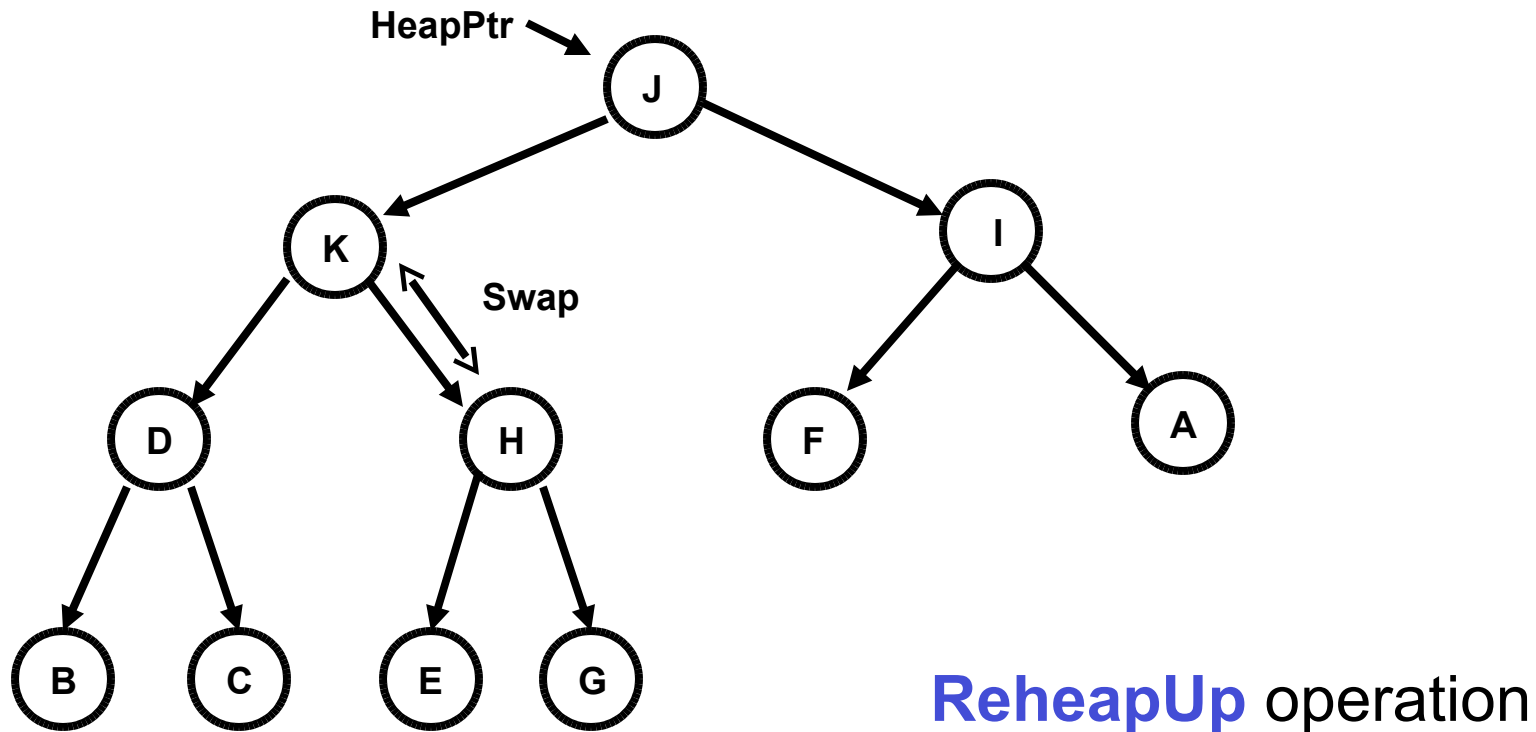
Order property is not maintained

Order must be adjusted

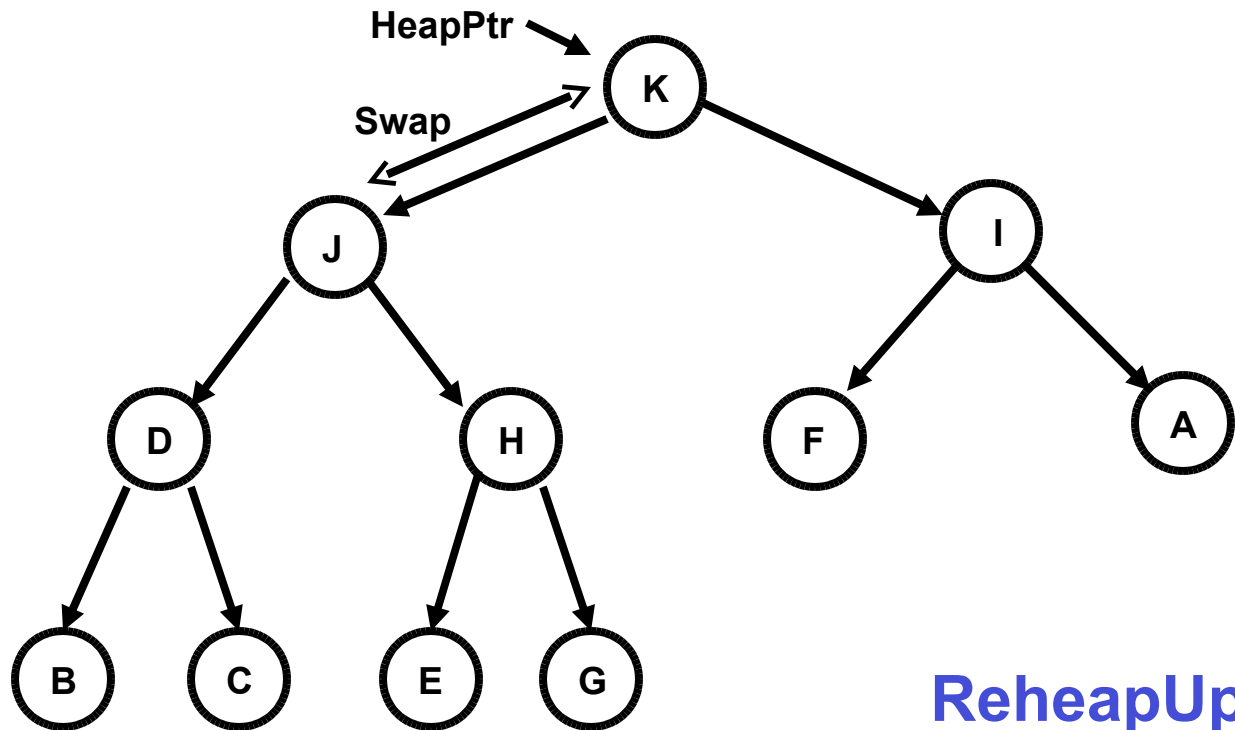
# Heap Example



# Heap Example



# Heap Example



**ReheapUp** operation

# Heap ADT

- ReheapUp(root,bottom)
  - Restores the order property of the heap between root and bottom
  - Precondition
    - The order property is satisfied from the root of the heap through the next-to-last node; the last (bottom) node may violate the order property
  - Postcondition
    - The order property applies to all elements of the heap

# Heap Implementation - 1

```
typedef int ItemType;

template<class ItemType>
// Assumes ItemType is either a built-in simple type or a class
// with overloaded relational operators.
struct HeapType
{
    void ReheapDown(int root, int bottom);
    void ReheapUp(int root, int bottom);
    ItemType* elements;    // Array to be allocated dynamically
    int numElements;
};
```

# Heap Implementation - 2

```
template<class ItemType>
void HeapType<ItemType>::ReheapUp(int root, int bottom)
// Post: Heap property is restored.
{
    int parent;

    if (bottom > root)
    {
        parent = (bottom-1) / 2;
        if (elements[parent] < elements[bottom])
        {
            Swap(elements[parent], elements[bottom]);
            ReheapUp(root, parent);
        }
    }
}
```



# Heap Implementation - 3

```
template<class ItemType>
void HeapType<ItemType>::ReheapDown(int root, int bottom)
// Post: Heap property is restored.
{
    int maxChild;
    int rightChild;
    int leftChild;

    leftChild = root*2+1;
    rightChild = root*2+2;
    if (leftChild <= bottom)
    {
        if (leftChild == bottom)
            maxChild = leftChild;
        else
        {
            if (elements[leftChild] <= elements[rightChild])
                maxChild = rightChild;
            else
                maxChild = leftChild;
        }
        if (elements[root] < elements[maxChild])
        {
            Swap(elements[root], elements[maxChild]);
            ReheapDown(maxChild, bottom);
        }
    }
}
```

# Heap Efficiency

- Insert Item
  - $O(\log n)$
- Remove maximum
  - $O(\log n)$
- Access maximum
  - $O(1)$

# Searching

- We have seen two techniques for determining if a particular element is stored in a data structure
  - Brute-force linear search  $O(N)$
  - Binary search  $O(\log_2 N)$

# Hashing

- *Hashing*
  - A technique for ordering and accessing list elements which can permit  $O(1)$  searching
  - Key value is used to place an item into the list at a particular location
  - Key also enables access to that location later
  - Trades memory space for access speed

# Hash Function

- A function used to manipulate the key of an element in a list to identify its location in the list
- Hash functions have two purposes
  - Used to access a particular item
  - Determines where to place the item
- Collision
  - The condition resulting when two or more keys produce the same hash location

# Hashing Example - 1

- Suppose one wishes to store employee records in an array
- Currently there are 100 employees
- Each student has a unique employee ID number in the range 00000-99999
- Employee ID numbers are not necessarily contiguous

# Hashing Example - 2

- **Option #1**
  - Declare an array of 100,000 records
  - Use the unique employee ID number directly as an array index
- Analysis
  - Wasteful, only 100 records needed but memory is reserved for 100,000
  - Provides  $O(1)$  access

# Hashing Example - 3

- **Option #2**
  - Declare an array just large enough to hold 100 records ==> valid indices 0-99
  - Use a Hashing Function to transform the employee ID number to a particular array index in the range 0-99
  - Consider the following hash function  
$$\text{ID} \% 100$$



# Hashing Example - 4

Examples:

–Employee #53330

$$53330 \% 100 = 30$$

–Employee #81235

$$81235 \% 100 = 35$$

	...
[30]	Employee 53330
[31]	Empty
[32]	Empty
[33]	Empty
[34]	Empty
[35]	Employee 53335
[36]	Empty
[37]	Empty
	...

# Hashing Example - 5

```
int ItemType::Hash() const
// NOTE: Hash function return value is a member of ItemType
// Post: Returns an integer between 0 and MAX_ITEMS -1.
{
    return (idNum % MAX_ITEMS); // idNum is a member of ItemType
}
```

```
template<class ItemType>
void ListType<ItemType>::InsertItem(ItemType item)
// Post: item is stored in the array at position item.Hash().
{
    int location;

    location = item.Hash();
    info[location] = item;
    length++;
}
```

```
template<class ItemType>
void ListType<ItemType>::RetrieveItem(ItemType& item)
// Post: Returns the element in the array at position
//       item.Hash().
{
    int location;

    location = item.Hash();
    item = info[location];
}
```

# Hashing Example - 6

Examples:

–Employee #53330

$$53330 \% 100 = 30$$

–Employee #81235

$$81235 \% 100 = 35$$

–Employee #10935

$$10935 \% 100 = 35 \text{ Collision!!}$$

Bin 35 is occupied but the next bin is empty so use it.

	...
[30]	Employee 53330
[31]	Empty
[32]	Empty
[33]	Empty
[34]	Empty
[35]	Employee 53335
[36]	Employee 10935
[37]	Empty
	...

# Hashing Example - 7

## Examples:

–Employee #53330

$$53330 \% 100 = 30$$

–Employee #81235

$$81235 \% 100 = 35$$

–Employee #10935

$$10935 \% 100 = 35 \text{ Collision!!}$$

–Employee #00035

$$00035 \% 100 = 35 \text{ Another Collision!!}$$

What happens if an employee ID hashes to the last bin in the array but it is occupied?

	...
[30]	Employee 53330
[31]	Empty
[32]	Empty
[33]	Empty
[34]	Empty
[35]	Employee 53335
[36]	Employee 10935
[37]	Employee 00035
	...

# Hashing with Collision Resolution

```
int ItemType::Hash() const
// Post: Returns an integer between 0 and MAX_ITEMS -1.
{
    return (idNum % MAX_ITEMS);
}

template<class ItemType>
void ListType<ItemType>::InsertItem(ItemType item)
// Post: item is stored in the array at position item.Hash()
//       or the next free spot.
{
    int location;

    location = item.Hash();
    while (info[location] != emptyItem)
        location = (location + 1) % MAX_ITEMS;
    info[location] = item;
    length++;
}

template<class ItemType>
void ListType<ItemType>::RetrieveItem(ItemType& item, bool& found)
{
    int location;
    int startLoc;
    bool moreToSearch = true;

    startLoc = item.Hash();
    location = startLoc;
    do
    {
        if (info[location] == item || info[location] == emptyItem)
            moreToSearch = false;
        else
            location = (location + 1) % MAX_ITEMS;
    } while (location != startLoc && moreToSearch);
    found = (info[location] == item);
    if (found)
        item = info[location];
}
```

# Deletions

- How does one delete an item from the array if linear probing has been used to resolve collisions?

# Collision Resolution - 1

- ***Linear Probing***
  - A technique which resolves a hash collision by sequentially searching a hash table for an empty location starting at the location returned by the hash function
- ***Clustering***
  - The tendency of elements to become unevenly distributed in the hash table, creating clusters about hash locations

# Collision Resolution - 2

- **Rehashing**

- Resolving a collision by computing a new hash location from a hash function that manipulates the original location rather than the element's key

## **Linear Probing**

Rehash function =  $(\text{hash value} + 1) \% 100$

Could also use  $(\text{hash value} + \text{constant}) \% 100$

Note: constant and array\_size should be relatively prime  
[successive rehashes will eventually cover every array index]



# Collision Resolution - 3

- **Quadratic Probing**

- Resolving has collisions by using the rehashing formula

$$(\text{HashValue} + J^2) \% \text{ArraySize}$$

where J is the number of times that the rehash function has been applied

- **Buckets**

- A collection of elements associated with a particular hash location

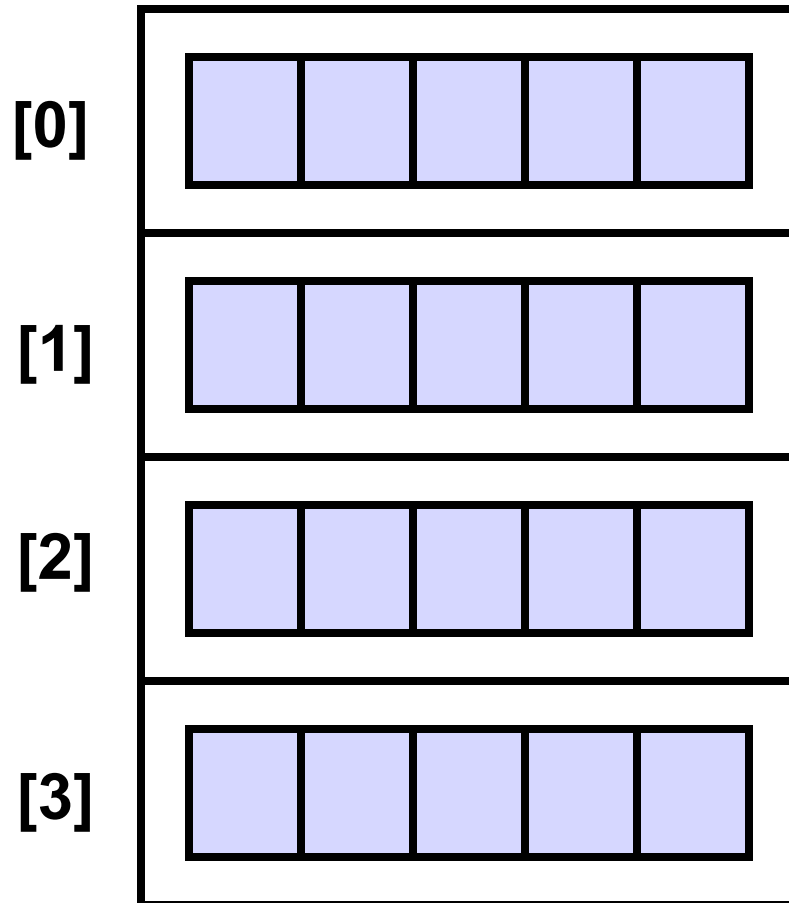
- **Chains**

- A linked list of elements that share the same hash location

# Buckets

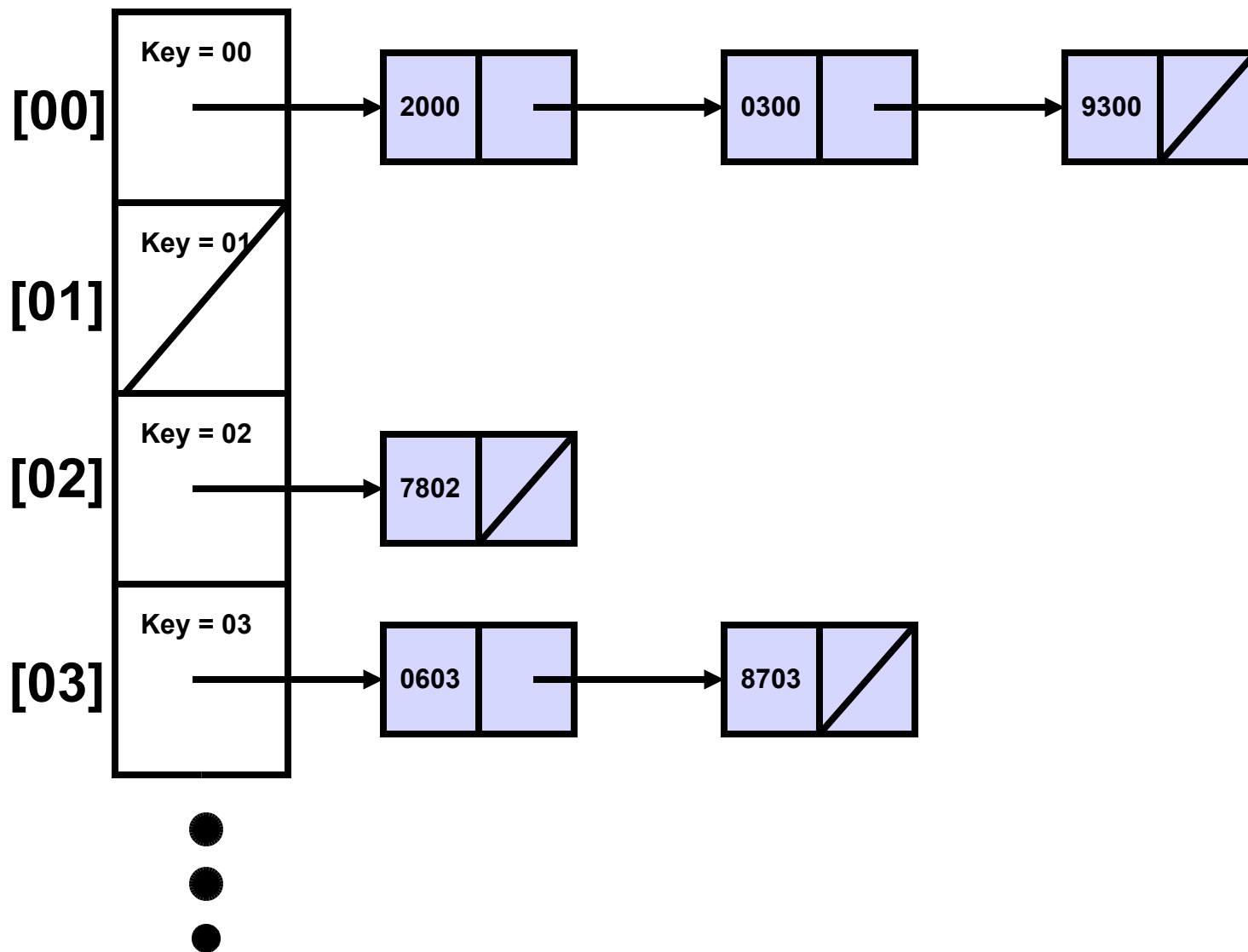
- Each hash location can hold an array of element values
- A collision causes the new item to be stored in one of the other local array bins
- If the local array is already full, something else must be done to resolve the collision

## buckets



# Chains

- Hash value provides an index into an array of pointers
- At a given location, the pointer is a link to a linked list which stores all items which hashed to the base location
- Eliminates collision resolution
- Simplifies deletions



# Designing Hash Functions

- Goals
  - Uniformly distribute items across array
  - Minimize collisions
- Need to know something about the distribution of the keys
- Division Method is most common
  - $\text{Key} \% \text{TableSize}$

# Folding

- A hash method that breaks the key into several pieces and concatenates or XORs some of the pieces to form the hash value

# What if your key is a string??

- Characters are represented internally by integers
- Can convert one or more characters of the string to a numeric value using the equivalent ASCII codes
- Can then use Division Method



# Load Factor

- For  $n$  items stored in an array of size  $N$ , the **Load Factor** is  $\frac{n}{N}$
- When the load factor approaches 1, the probability of a collision also approaches 1