SORTING

Bubble Sort

- Starting with the first element(index = 0), compare the current element with the next element of the array.
- If the current element is greater than the next element of the array, swap them.
- If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Bubble Sort [cont..]

- BUBBLE_SORT(A, N)

  Step 1: Repeat Step 2 For i = 0 to N-1

  Step 2: Repeat For j = 0 to N - I- i

  Step 3: IF A[j] > A[j + 1]

  SWAP A[j] and A[j+1]

  [END OF INNER LOOP]

  [END OF OUTER LOOP]

  Step 4: EXIT

Complexity of Bubble Sort

- In Bubble Sort, ***n-1*** comparisons will be done in the 1st pass, ***n-2*** in 2nd pass, ***n-3*** in 3rd pass and so on. So the total number of comparisons will be,

f(n) = (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1

f(n) = n(n-1)/2

i.e O(n^2)

Selection Sort

- algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire array is sorted.
- It is called selection sort because it repeatedly **selects** the next-smallest element and swaps it into the right place.

How Selection Sort works?

- Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
- We then move on to the second position, and look for smallest element present in the subarray, starting from index 1, till the last index.
- We replace the element at the **second** position in the original array, or we can say at the first position in the subarray, with the second smallest element.

Selection Sort

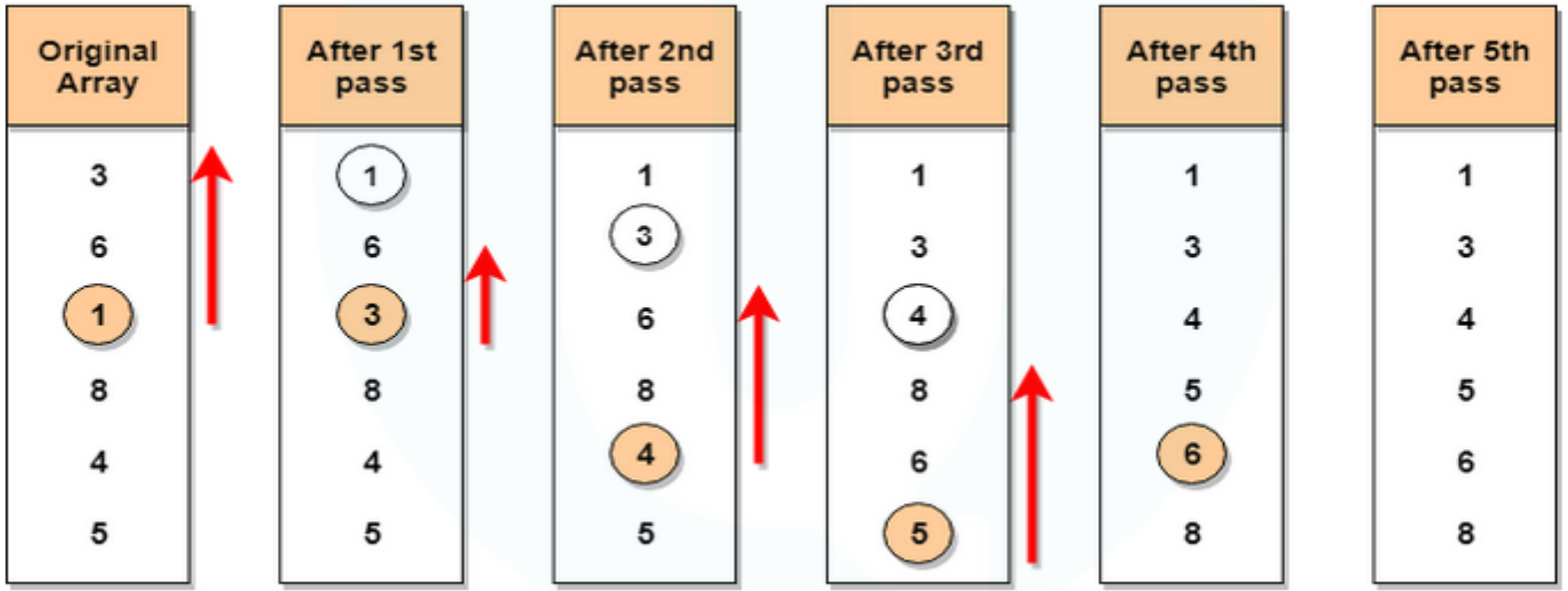
SMALLEST (ARR, K, N, POS)

```
Step 1: [INITIALIZE]
 SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N -1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
[END OF LOOP]
Step 4: RETURN POS
```

SELECTION SORT(ARR, N)

```
Step 1: Repeat Steps 2 and 3 for K = 0 to N-1
Step 2: CALL SMALLEST(ARR, K, N,POS)
      Step 3: SWAP A[K] with ARR[POS]
                  [END OF LOOP]
      Step 4: Exit
```

Example



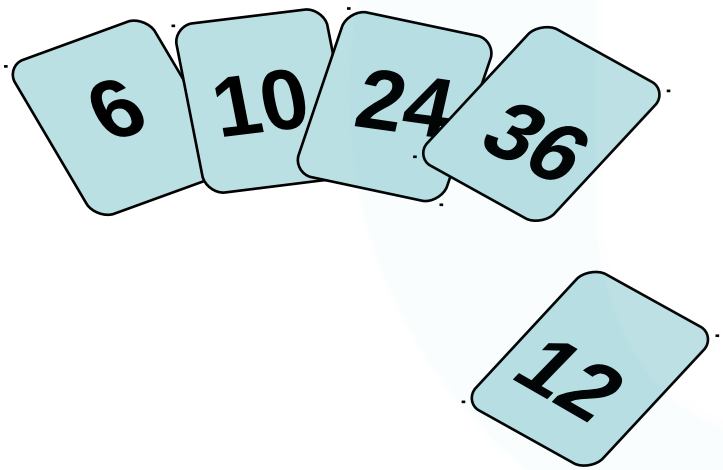
Complexity of Selection Sort

- In Pass 1, selecting the element with the smallest value calls for scanning all n elements;thus, n–1 comparisons are required in the first pass.
- Then, the smallest value is swapped with the element in the first position. In Pass 2, selecting the second smallest value requires scanning the remaining n – 1 elements and so on. Therefore,
- f(n) = (n – 1) + (n – 2) + ... + 2 + 1
- f(n) = n(n – 1) / 2 = $O(n^2)$ comparisons

Insertion Sort

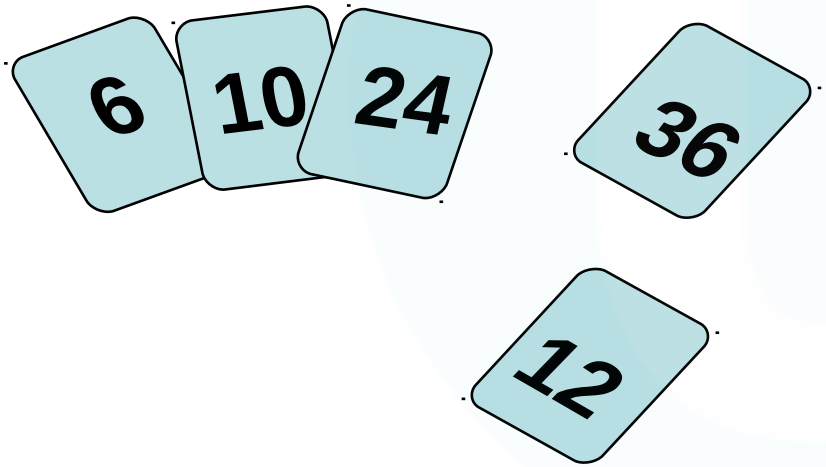
- Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands

Insertion Sort

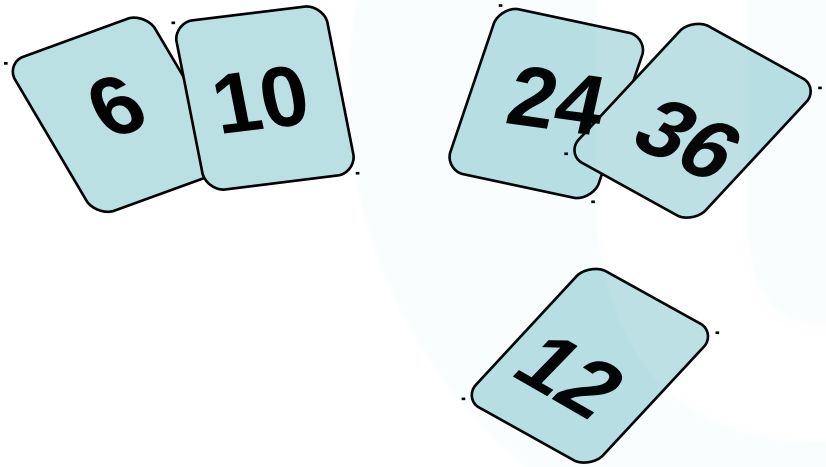


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



Insertion Sort

input array

5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:

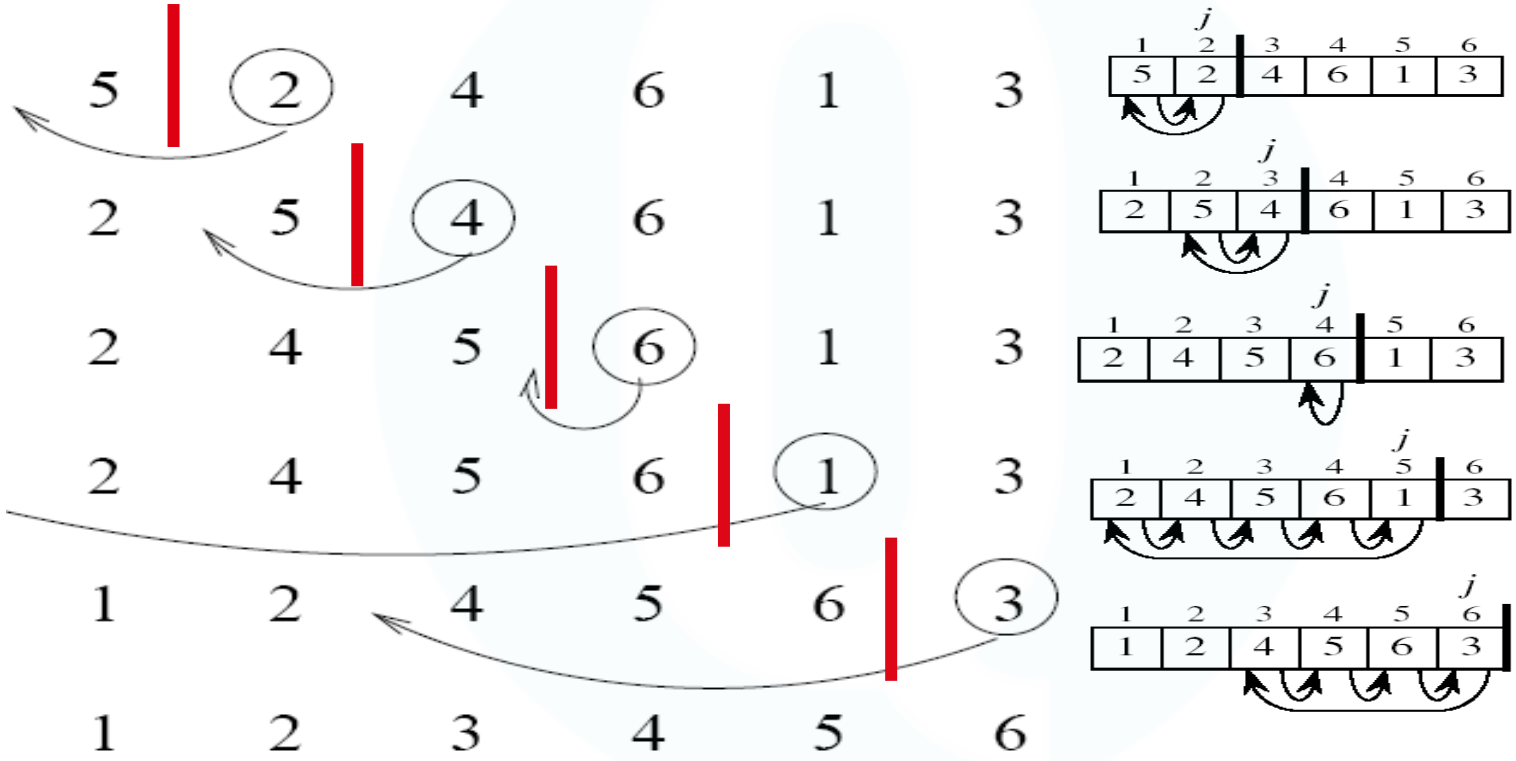
left sub-array

right sub-array

sorted

unsorted

Insertion Sort



INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N – 1

Step 2: SET TEMP = ARR[K]

Step 3: SET J = K - 1

Step 4: Repeat while TEMP <= ARR[J] and J>=0
SET ARR[J + 1] = ARR[J]
SET J = J - 1
[END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP

[END OF LOOP]

Step 6: EXIT

Complexity of Insertion Sort

- The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., O(n)).
- During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.
- The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n2)).
- The average case is also quadratic

Quick Sort

The quick sort algorithm works as follows:

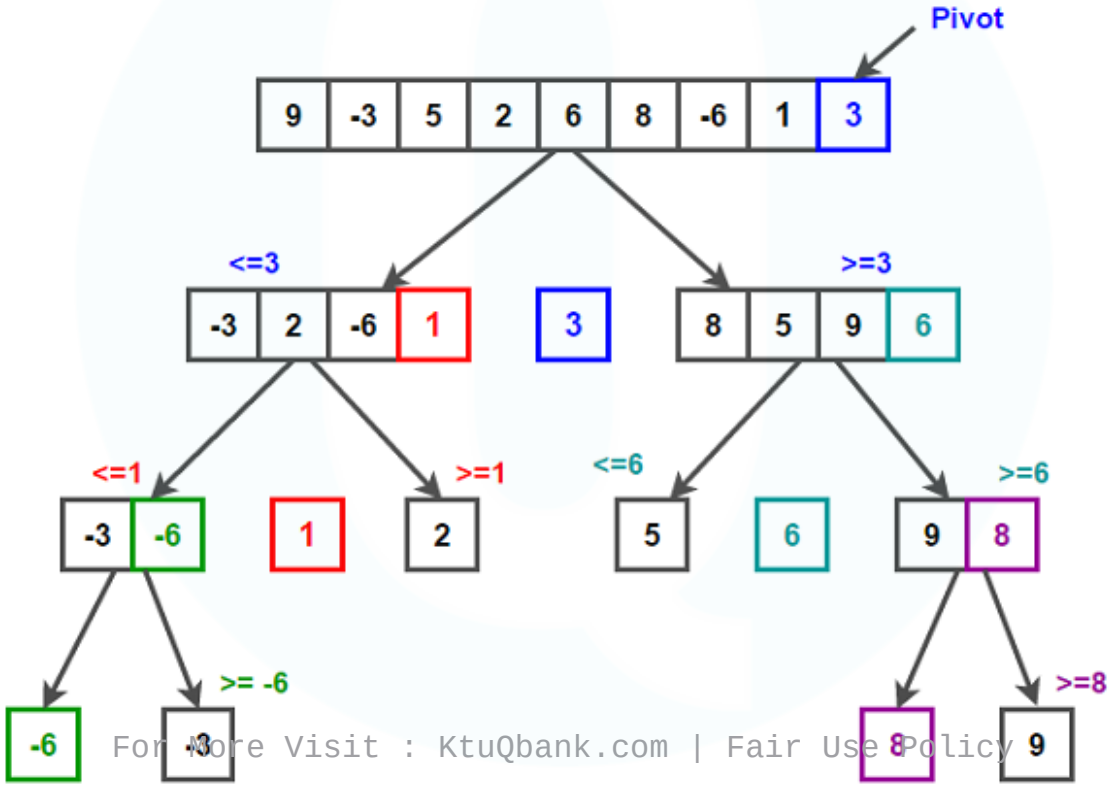
1. Select an element pivot from the array elements.

2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position.

This is called the partition operation.

3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

Quick Sort



Quick Sort

QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)

Q = PARTITION (ARR, P , R)

QUICKSORT(ARR, P, Q-1)

QUICKSORT(ARR, Q+1, R)

[END OF IF]

Step 2: END

PARTITION (ARR, P, R)

```
{
    X = ARR[R]
    i = P - 1
    For j = P to R - 1
    {
        If ( ARR[ j ] <= X )
        {
            i = i + 1
            Exchange ARR[ i ] with ARR[ j ]
        }
    }
    Exchange ARR[ i + 1] with ARR[ R ]
    Return i + 1
}
```

Complexity of Quick Sort

- In the best case, every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size. At the most, only log n nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is O(log n) . And because at each level, there can only be O(n) , the resultant time is given as **O(nlog n)** time.

Complexity

- Its worst-case efficiency is given as O(n 2) . The worst case occurs when the array is already sorted (either in ascending or descending order) and the left-most element is chosen as the pivot

Merge Sort

- Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.
- ***Divide*** means partitioning the n -element array to be sorted into two sub-arrays of n/2 elements. If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A_1 and A_2 , each containing about half of the elements of A .
- ***Conquer*** means sorting the two sub-arrays recursively using merge sort.
- ***Combine*** means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

Algorithm for Merge Sort

MERGE_SORT(ARR, P, R)

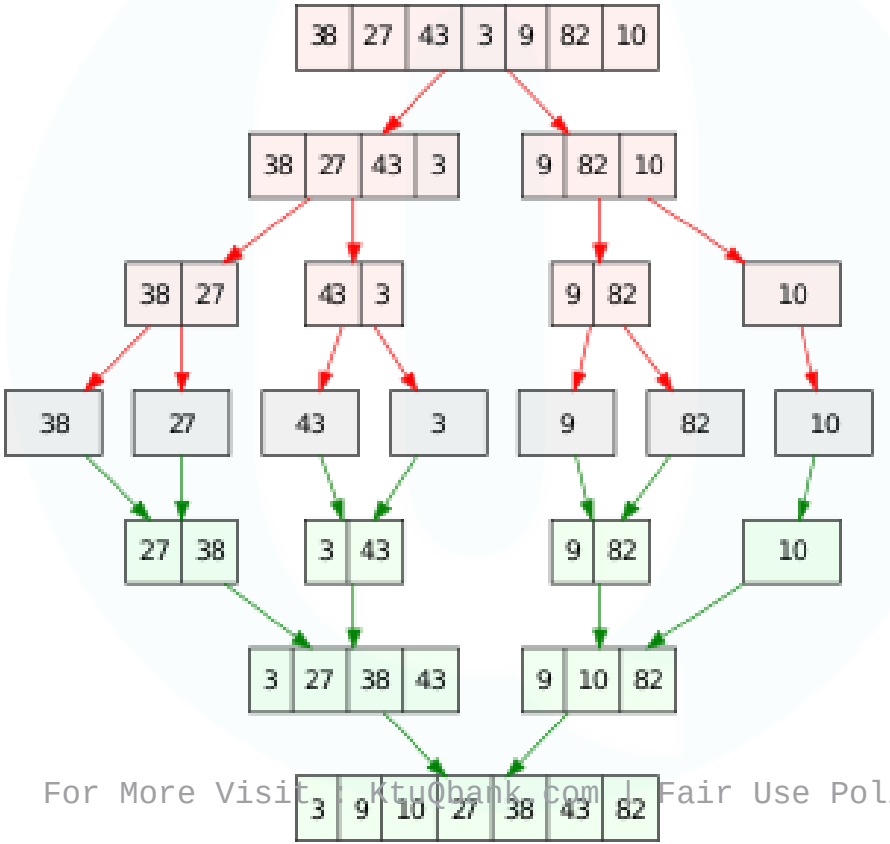
```
mergesort(p,q)
if(p<q)     // if there exist more than on element
    m=(p+q)/2    // Dividing
    mergesort(p,m)
    mergesort(m+1,q)
    merge(p,m,q) // Merging
End
```

Algorithm for Merge Sort[cont...]

MERGE (ARR, P, Q, R)

```
merge(low,m,high)
i=low
j=m+1
k=low
while(i<=m&&j<=high) // comparing
                                elements
    if(a[i]<=a[j])
    b[k++]=a[i++]
    else
    b[k++]=a[j++]
```

```
if(i>m) // Finished 1st half and copying
            // the remaining nos of 2nd half
    for(p=j; p<=high; p++)
        b[k++]=a[p]
else     // Finished 2nd half and copying
          // the remaining nos of 1st half
    for(p=i; p<=m; p++)
       b[k++]=a[p]
for(p=low; p<=high; p++)
      a[p]=b[p]
        // copy the elements of array b to a
}
```



Complexity of Merge Sort

The list of size **N** is divided into a max of **log N** parts, and the merging of all sublists into a single list takes **O(N)** time, the worst case run time of this algorithm is **O(N log N)**

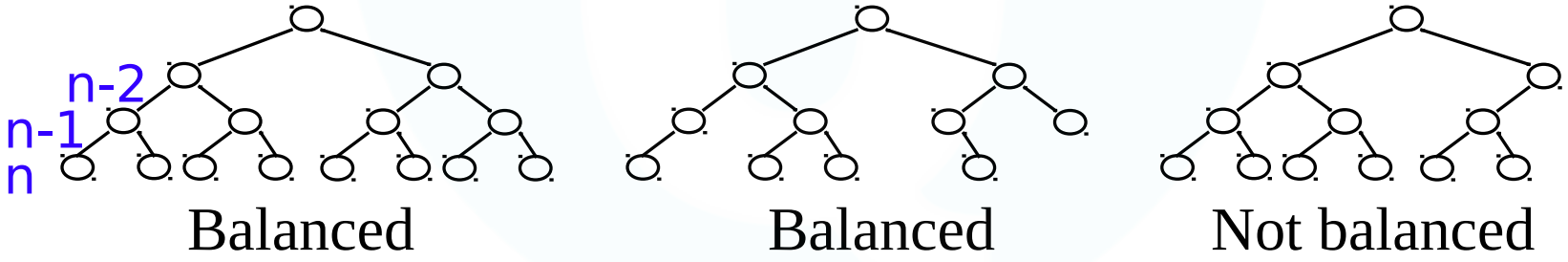
Heapsort

Heap data structure

- Binary tree
- Balanced
- Left-justified or Complete
- (Max) Heap property: no node has a value greater than the value in its parent

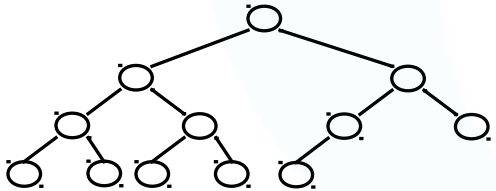
Balanced binary trees

- Recall:
  - The depth of a node is its distance from the root
  - The depth of a tree is the depth of the deepest node
- A binary tree of depth n is balanced if all the nodes at depths 0 through n-2 have two children

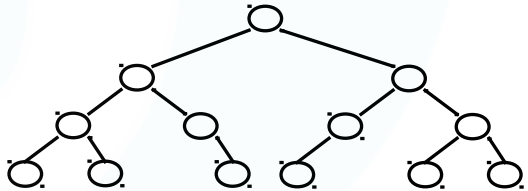


Left-justified binary trees

- A balanced binary tree of depth n is left-justified if:
  - it has 2^n nodes at depth n (the tree is “full”), or
  - it has 2^k nodes at depth k, for all $k < n$, *and* all the leaves at depth n are as far left as possible



Left-justified



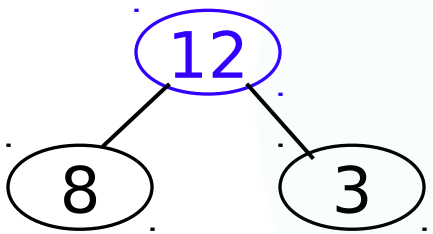
Not left-justified

Building up to heap sort

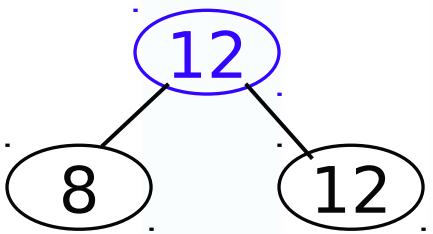
- How to build a heap
- How to maintain a heap
- How to use a heap to sort data

The heap property

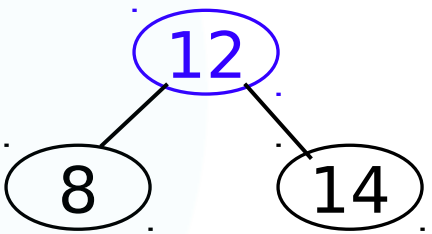
- A node has the heap property if the value in the node is as large as or larger than the values in its children



Blue node has heap property



Blue node has heap property

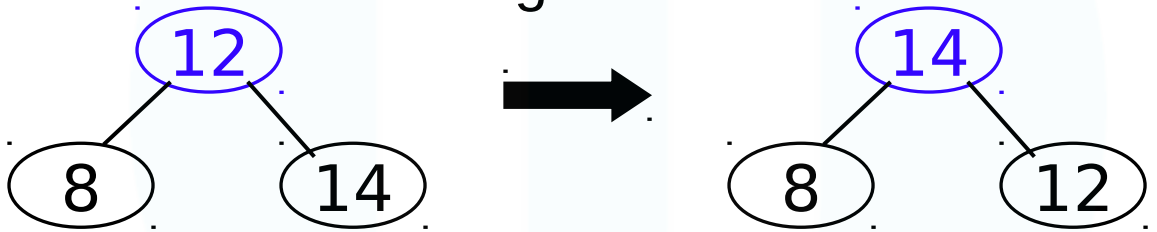


Blue node does not have heap property

- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

siftUp

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



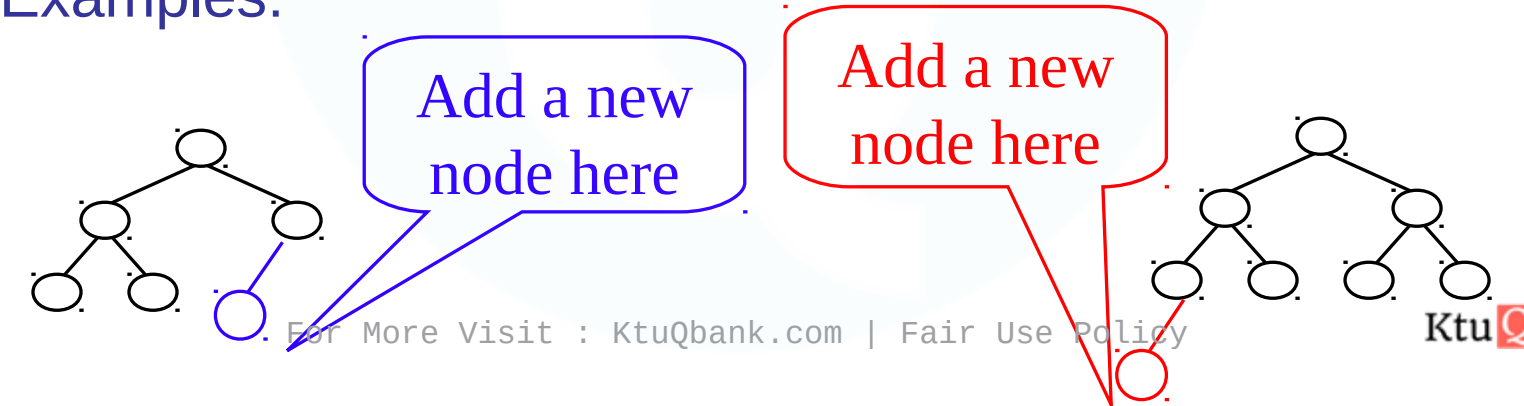
Blue node does not have heap property

Blue node has heap property

- This is sometimes called sifting up

Constructing a heap I

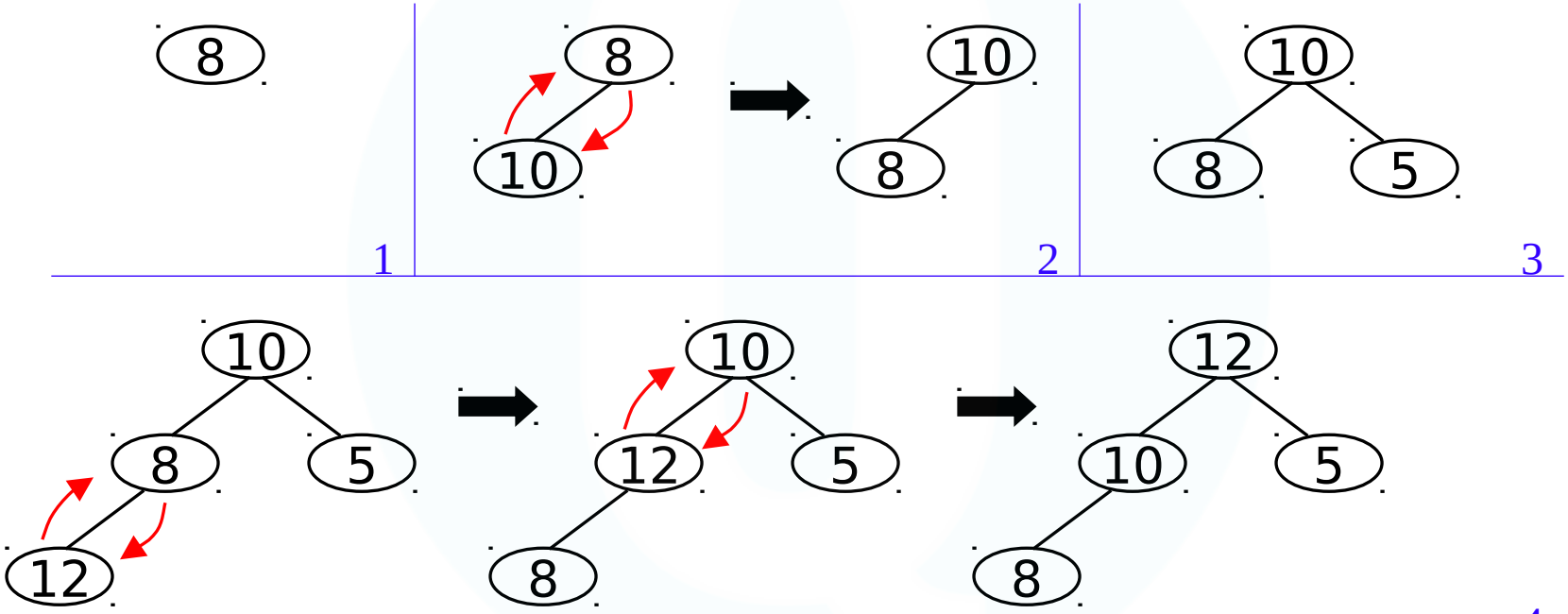
- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level
  - If the deepest level is full, start a new level
- Examples:



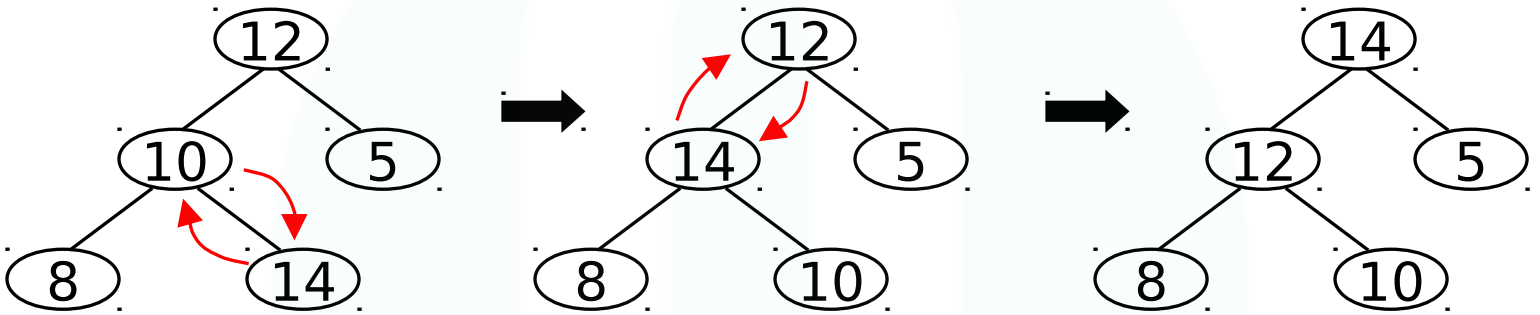
Constructing a heap II

- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we sift up
- But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of *its* parent node
- We repeat the sifting up process, moving up in the tree, until either
  - We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
  - We reach the root

Constructing a heap III



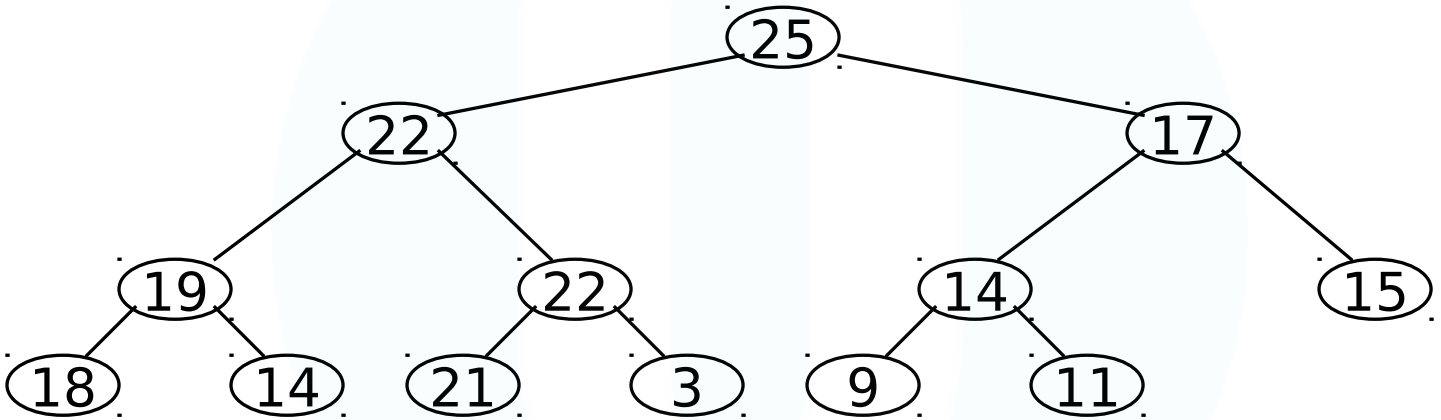
Other children are not affected



- The node containing 8 is not affected because its parent gets larger, not smaller
- The node containing 5 is not affected because its parent gets larger, not smaller
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A sample heap

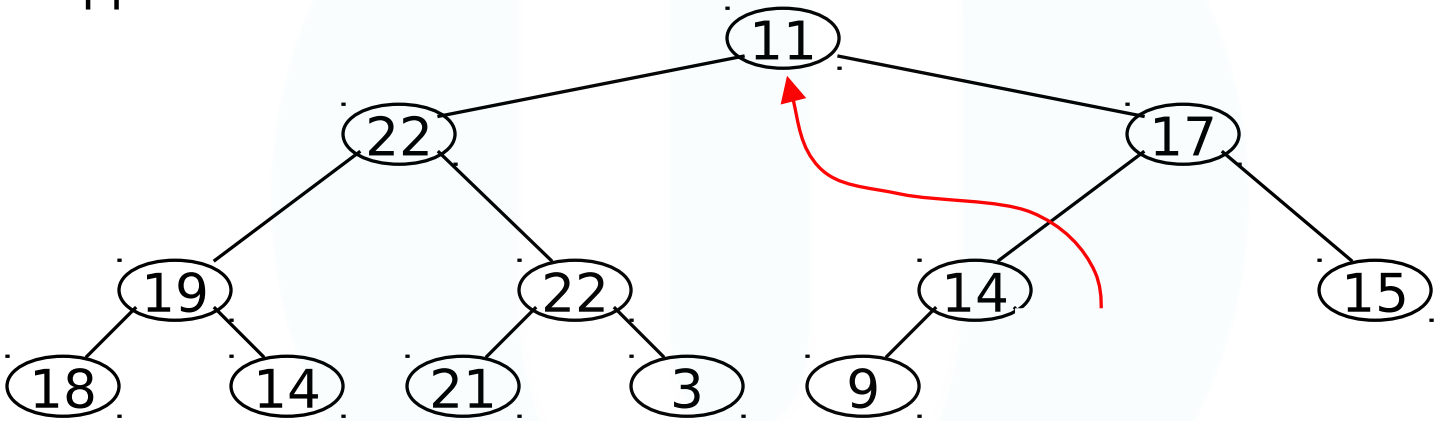
- Here's a sample binary tree after it has been heapified



- Notice that heapified does *not* mean sorted
- Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

Removing the root (animated)

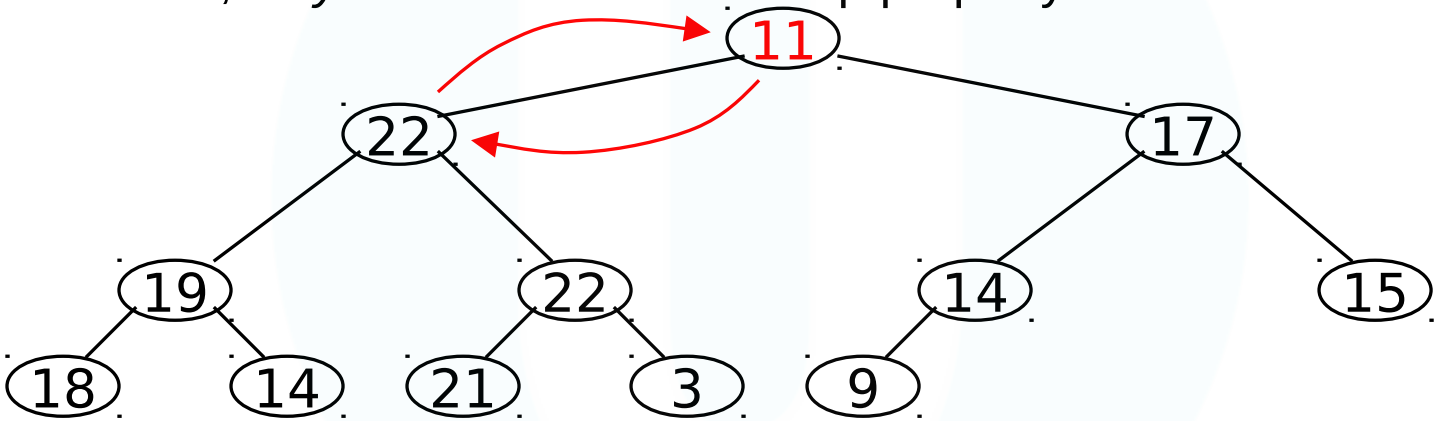
- Notice that the largest number is now in the root
- Suppose we *discard* the root:



- How can we fix the binary tree so it is once again *balanced and left-justified?*
- Solution: remove the rightmost leaf at the deepest level and use it for the new root

The reHeap method I

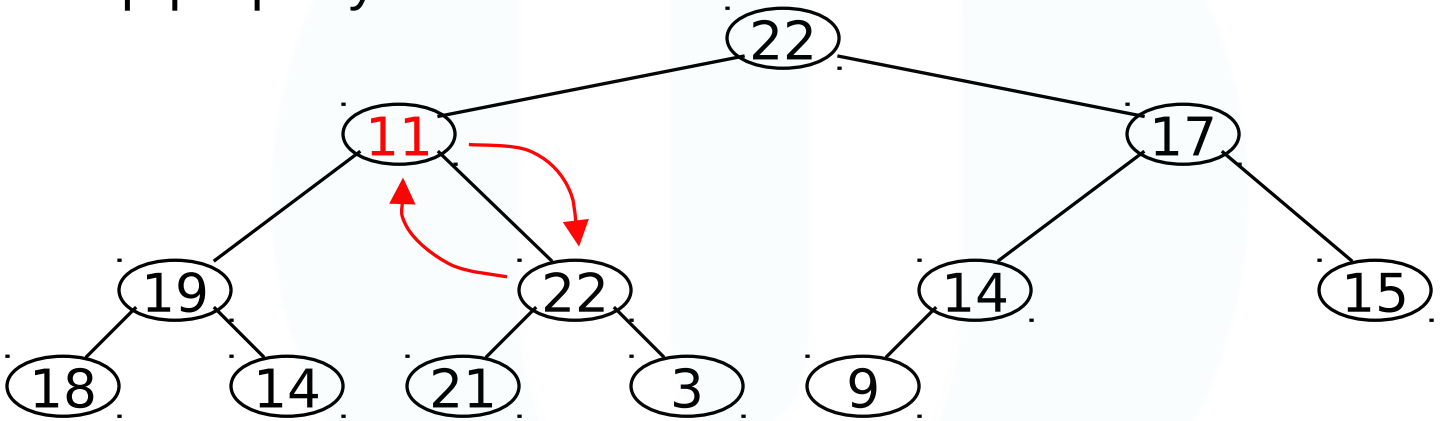
- Our tree is balanced and left-justified, but no longer a heap
- However, *only the root* lacks the heap property



- We can siftDown() the root
- After doing this, one and only one of its children may have lost the heap property

The reHeap method II

- Now the left child of the root (still the number 11) lacks the heap property

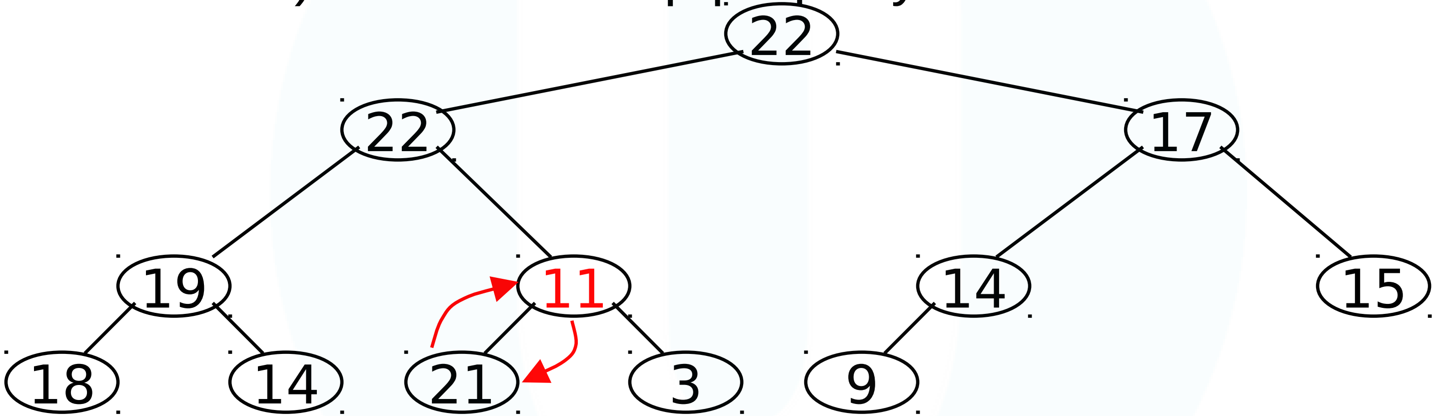


- We can siftDown() this node
- After doing this, one and only one of its children may have lost the heap property

Ktu Q bank

The reHeap method III

- Now the right child of the left child of the root (still the number 11) lacks the heap property:

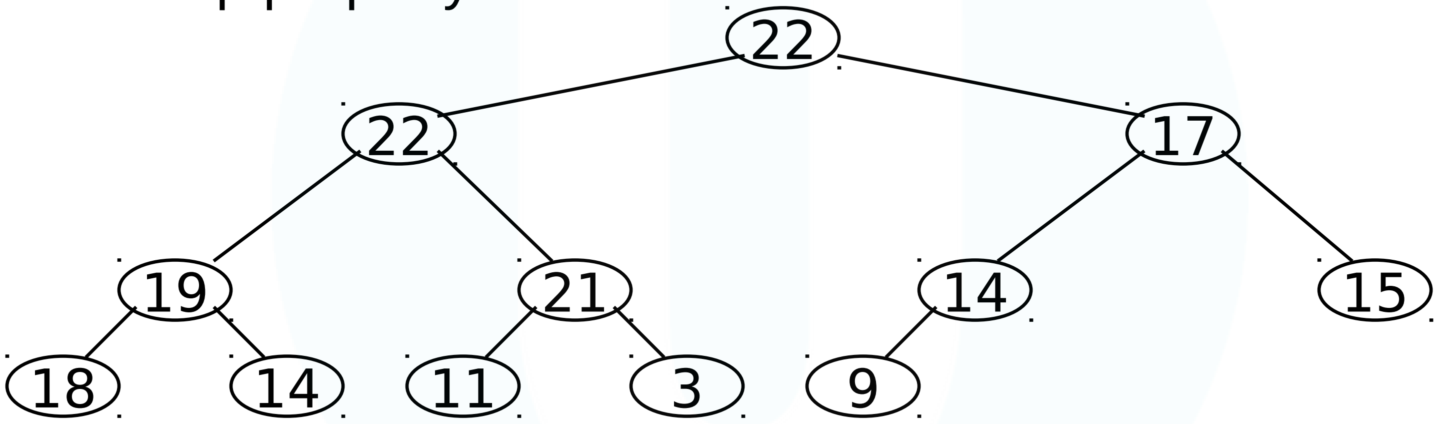


- We can siftDown() this node
- After doing this, one and only one of its children may have lost the heap property—but it doesn't, because it's a leaf

Ktu Q bank

The reHeap method IV

- Our tree is once again a heap, because every node in it has the heap property



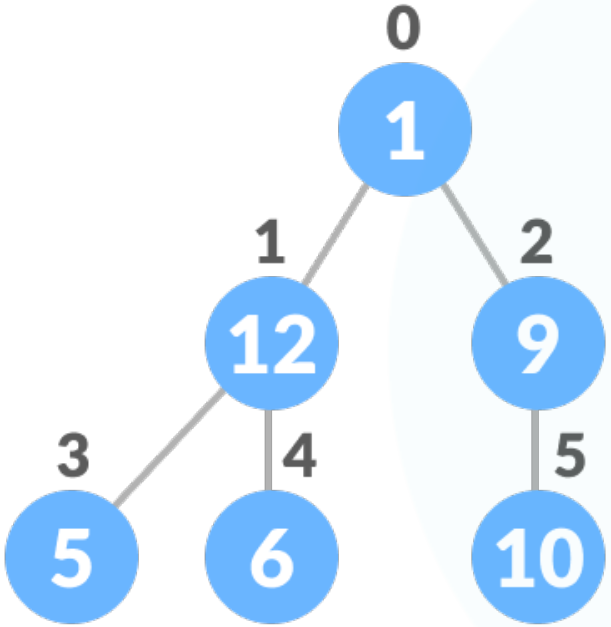
- Once again, the largest (or *a* largest) value is in the root
- We can repeat this process until the tree becomes empty
- This produces a sequence of values in order largest to smallest

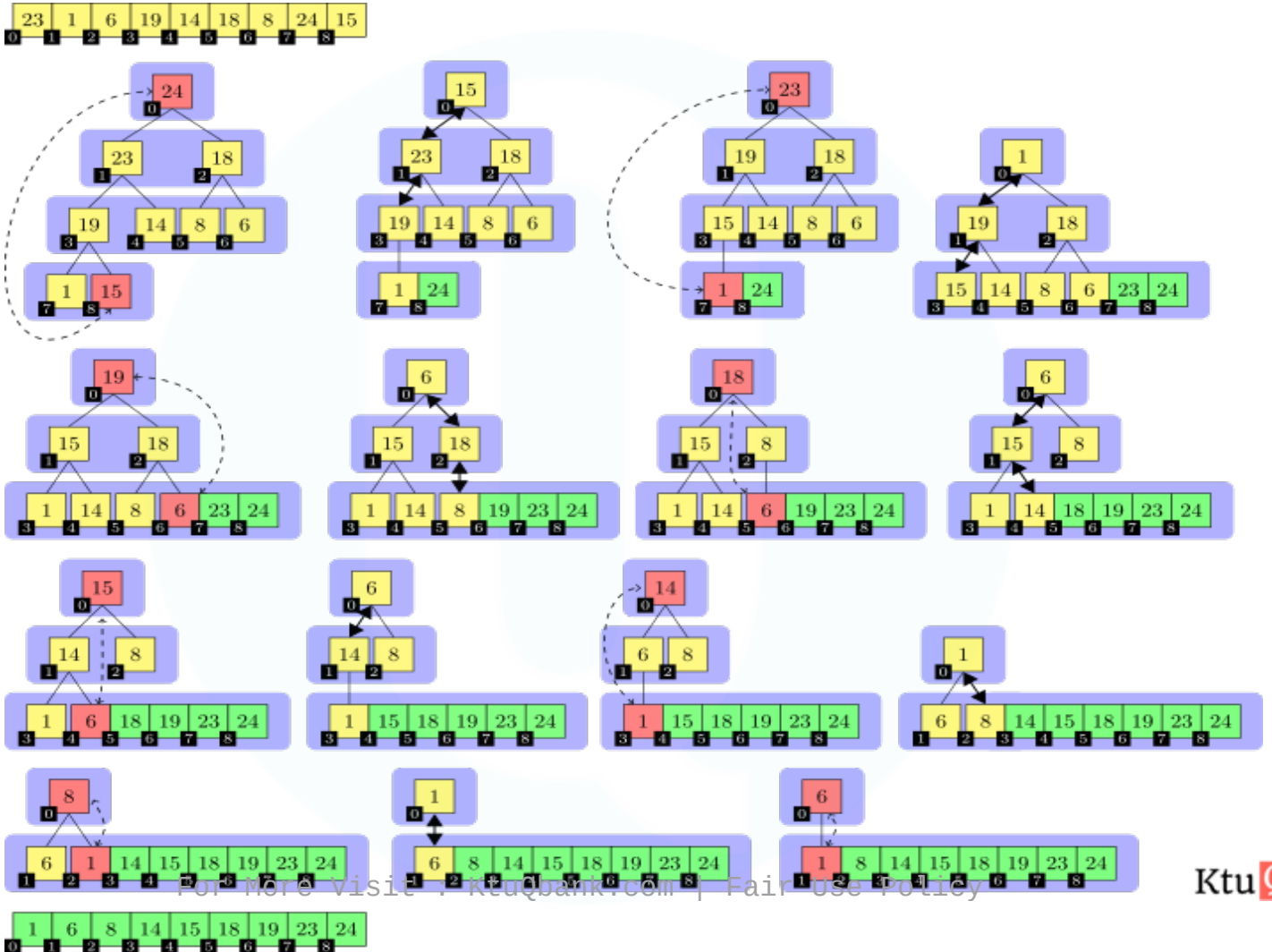
Sorting

- What do heaps have to do with sorting an array?
  - Because the binary tree is *balanced* and *left justified*, it can be represented as an array
  - All our operations on binary trees can be represented as operations on *arrays*
  - To sort:

```
heapify the array;
while the array isn't empty {
    remove and replace the root;
    reheap the new root node;
}
```

Sorting





Analysis

- To reheap the root node, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)
- The binary tree is perfectly balanced
- Therefore, this path is O(log n) long
  - And we only do O(1) operations at each node
  - Therefore, reheaping takes O(log n) times
- Since we reheap inside a while loop that we do n times, the total time for the while loop is n*O(log n), or O(n log n)

Analysis

- Construct the heap O(n log n)
- Remove and re-heap O(log n)
  - Do this n times O(n log n)
- Total time O(n log n) + O(n log n)

HASHING

KtuQbank

Introduction

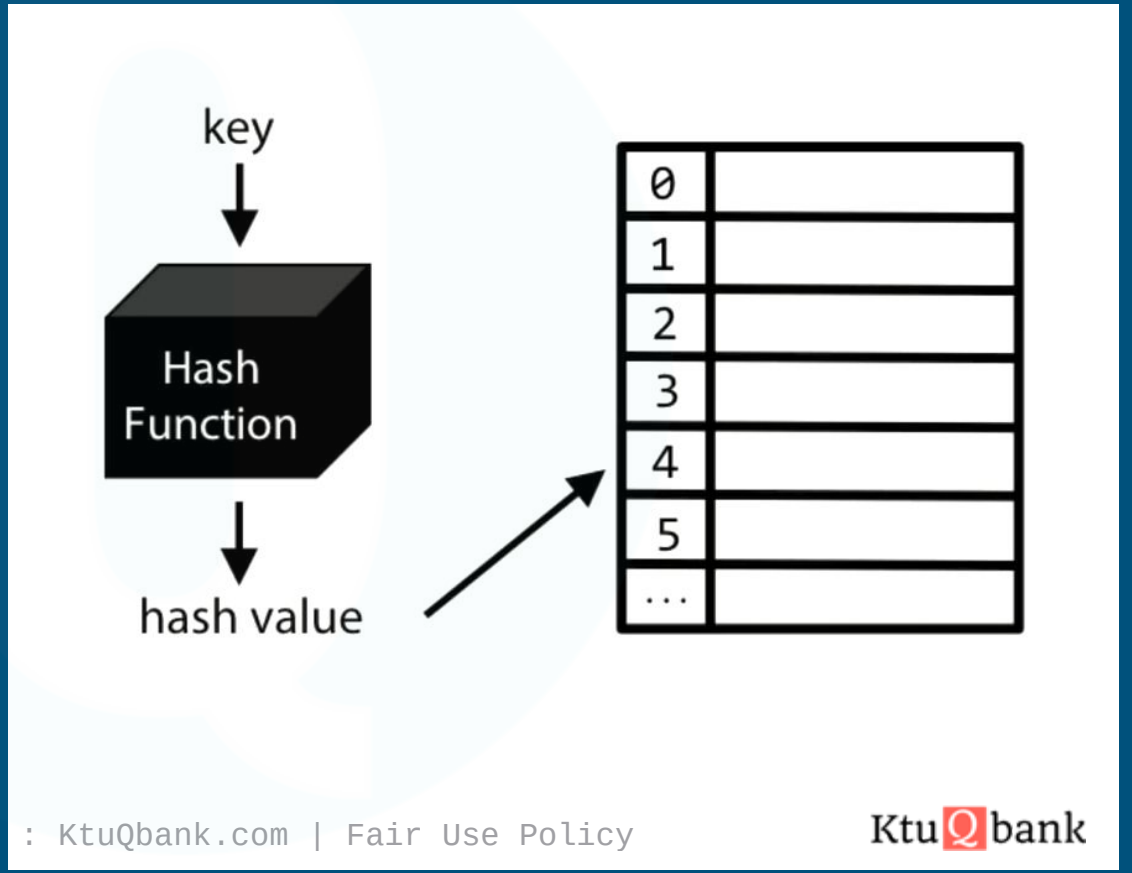
- Hashing is a method to store data in an array so that sorting, searching, inserting and deleting data is fast. For this every record needs unique key.
- The basic idea is not to search for the correct position of a record with comparisons but to compute the position within the array. The function that returns the position is called the 'hash function' and the array is called a 'hash table'

Introduction

- Sequential search requires, on the average O(n) comparisons to locate an element. So many comparisons are not desirable for a large database of elements.
- Binary search requires much fewer comparisons on the average O (log n) but there is an additional requirement that the data should be sorted. Even with best sorting algorithm, sorting of elements require 0(n log n) comparisons.
- There is another widely used technique for storing of data called hashing. It does away with the requirement of keeping data sorted (as in binary search) and its best case timing complexity is of constant order (0(1)). In its worst case, hashing algorithm starts behaving like linear search.
- Best case timing behavior of searching using hashing = O(1)
- Worst case timing Behavior of searching using hashing = O(n)
- In hashing, the record for a key value "key", is directly referred by calculating the address from the key value. Address or location of an element or record, x, is obtained by computing some arithmetic function f. f(key) gives the address of x in the table.

Hashing

- The function that returns the position is called **Hash Function** and the array is called a **Hash Table**



Different Hash Functions

Division Method

It is the most simple method of hashing an integer x . This method divides x by M and then uses the remainder obtained.

In this case, the hash function can be given as

$h(x) = x \bmod M$

Division Method

We can write,

L = (K mod m)

where L => location in table/file

K => key value

m => table size/number of slots in file

- Suppose, k = 23, m = 10 then

L = (23 mod 10) = 3 , The key whose value is 23 is placed in 3rd location.

Mid Square Method

The mid-square method is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k^2 .

Step 2: Extract the middle r digits of the result obtained in Step 1.

Mid Square Method

Suppose a key value is 16, then its square is 256. Now if we want address of two digits, then you select the address as 56 (i.e. two digits starting from middle of 256).

Folding Method

The folding method works in the following two steps:

Step 1: Divide the key value into a number of parts. That is, divide k into parts k_1 , k_2 , ... , k_n , where each part has the same number of digits except the last part which may have lesser digits than the other parts.

Step 2: Add the individual parts. That is, obtain the sum of $k_1 + k_2 + ... + k_n$. The hash value is produced by ignoring the last carry, if any.

Folding Method

- **This is done in two ways :**

Fold-shifting: Here actual values of each parts of key are added.

ƒ Suppose, the key is : 12345678, and the required address is of two digits,

ƒ Then break the key into: 12, 34, 56, 78.

ƒ Add these, we get 12 + 34 + 56 + 78 : 180, ignore first 1 we get 80 as location

Fold-boundary: Here the reversed values of outer parts of key are added.

ƒ Suppose, the key is : 12345678, and the required address is of two digits,

ƒ Then break the key into: 21, 34, 56, 87.

ƒ Add these, we get 21 + 34 + 56 + 87 : 198, ignore first 1 we get 98 as location

Digit Analysis Method

- This hashing function is a distribution-dependent.
- Here we make a statistical analysis of digits of the key, and select those digits which occur quite frequently.
- Then reverse or shifts the digits to get the address.
- For example, if the key is : 9861234. If the statistical analysis has revealed the fact that the third and fifth position digits occur quite frequently, then we choose the digits in these positions from the key.

So we get, 62. Reversing it we get 26 as the address.

Properties of a Good Hash Function

- ***Low cost:*** The cost of executing a hash function must be small, so that using the hashing technique becomes preferable over other approaches. For example, if binary search algorithm can search an element from a sorted table of n items with $\log_2 n$ key comparisons, then the hash function must cost less than performing $\log_2 n$ key comparisons.

Properties of a Good Hash Function

- ***Determinism:*** A hash procedure must be deterministic. This means that the same hash value must be generated for a given input value. However, this criteria excludes hash functions that depend on external variable parameters (such as the time of day) and on the memory address of the object being hashed (because address of the object may change during processing).

Properties of a Good Hash Function

Uniformity: A good hash function must map the keys as evenly as possible over its output range. This means that the probability of generating every hash value in the output range should roughly be the same. The property of uniformity also minimizes the number of collisions.

Collision

Collisions occur when the hash function maps two different keys to the same location. Obviously, two records cannot be stored in the same location. Therefore, a method used to solve the problem of collision, also called collision resolution technique, is applied.

The two most popular methods of resolving collisions are:

1. Open addressing

2. Chaining

Collision Resolution by Open Addressing / Closed Hashing

The hash table contains two types of values: sentinel values (e.g., –1) and data values. The presence of a sentinel value indicates that the location contains no data value at present but can be used to hold a value.

Collision Resolution by Open Addressing / Closed Hashing

When a key is mapped to a particular memory location, then the value it holds is checked. If it contains a sentinel value, then the location is free and the data value can be stored in it. However, if the location already has some data value stored in it, then other slots are examined systematically in the forward direction to find a free slot. If even a single free location is not found, then we have an OVERFLOW condition.

The process of examining memory locations in the hash table is called probing. Open addressing technique can be implemented using

- Linear probing
- Quadratic probing
- Double hashing
- Rehashing.

Linear Probing

In this technique, if a value is already stored at a location generated by h(k) , then the following hash function is used to resolve the collision:

h(k, i) = [h ‘(k) + i] mod m

Where m is the size of the hash table, h ‘ (k) = (k mod m) , and i is the probe number that varies from 0 to m–1 .

Quadratic Probing

In this technique, if a value is already stored at a location generated by h(k) , then the following hash function is used to resolve the collision:

$$\mathbf{h(k, i) = [h\,'(k) + c_1\, i + c_2\, i^2\,]\ mod\ m}$$

where m is the size of the hash table, h'(k) = (k mod m) , i is the probe number that varies from 0 to m - 1 and c_1 and c_2 are constants such that c_1 and c_2 != 0 .

Double Hashing

- Double hashing uses one hash value and then repeatedly steps forward an interval until an empty location is reached.
- The interval is decided using a second, independent hash function, hence the name double hashing.
- In double hashing, we use two hash functions rather than a single
- function.

Double Hashing

- The hash function in the case of double hashing can be given as:

  $$\mathbf{h(k, i) = [h_1(k) + ih_2(k)] \bmod m}$$

  where m is the size of the hash table, $h_1(k)$ and $h_2(k)$ are two hash functions given as $h_1(k) = k \bmod m$, $h_2(k) = k \bmod m'$,

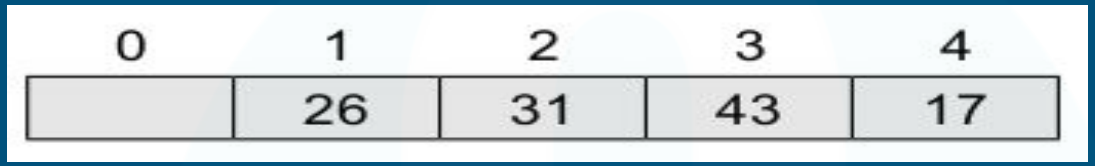
  i is the probe number that varies from 0 to m–1 , and m ' is chosen to be less than m . We can choose m ' = m–1 or m–2 .

Rehashing

When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. In such cases, a better option is to create a new hash table with size double of the original hash table.

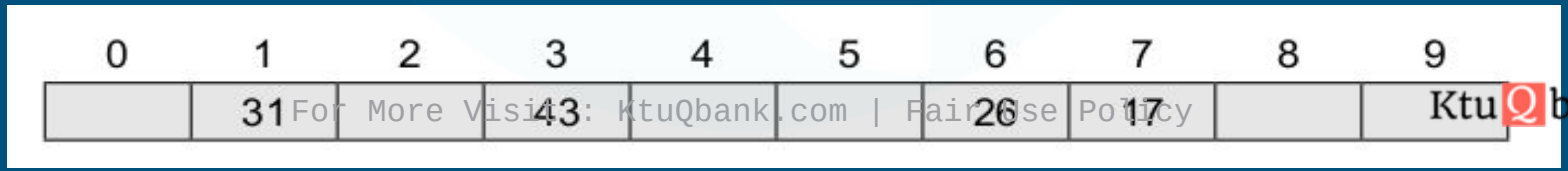
All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash value, and then inserting it in the new hash table.

Consider the hash table of size 5 given below. The hash function used is h(x) = x % 5 . Rehash the entries into to a new hash table.



Note that the new hash table is of 10 locations, double the size of the original table.

Now, rehash the key values from the old hash table into the new one using hash function— h(x) = x % 10 .



Collision Resolution by Chaining

- In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location.
- That is, location l in the hash table points to the head of the linked list of all the key values that hashed to l. However, if no key value hashes to l, then location l in the hash table contains NULL .

In chaining,

- Instead of a hash table, a table of linked list is used
- Keep a linked list of keys that hash to the same value.

Eg:

Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use **h(k) = k mod m .**

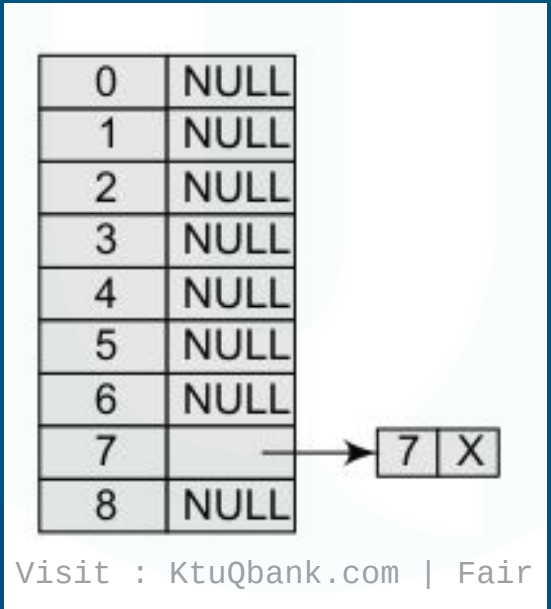
In this case, m=9 . Initially, the hash table can be given as:

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL

Step 1 : Key = 7

h(k) = 7 mod 9 = 7

Create a linked list for location 7 and store the key value 7 in it as its only node.

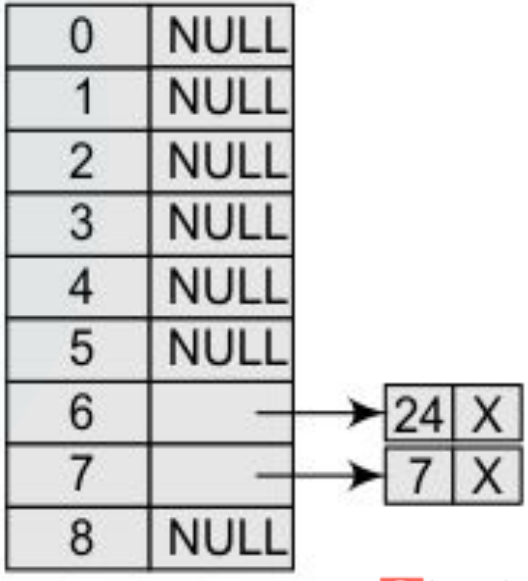


Step 2 : Key = 24

h(k) = 24 mod 9

= 6

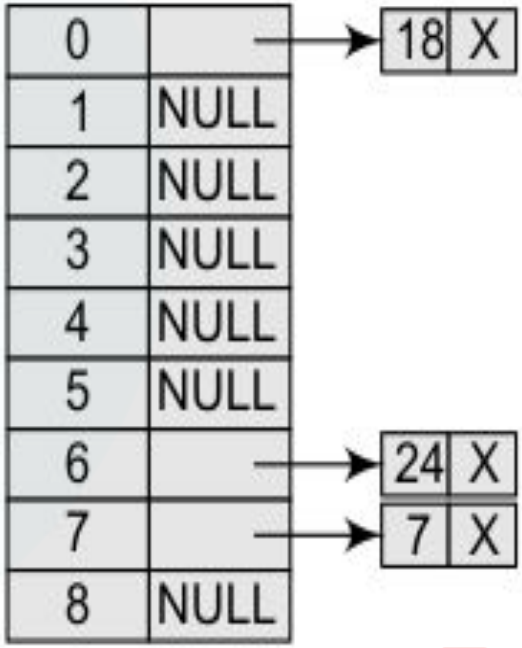
Create a linked list for location 6 and store the key value 24 in it as its only node.



Step 3 : Key = 18

h(k) = 18 mod 9 = 0

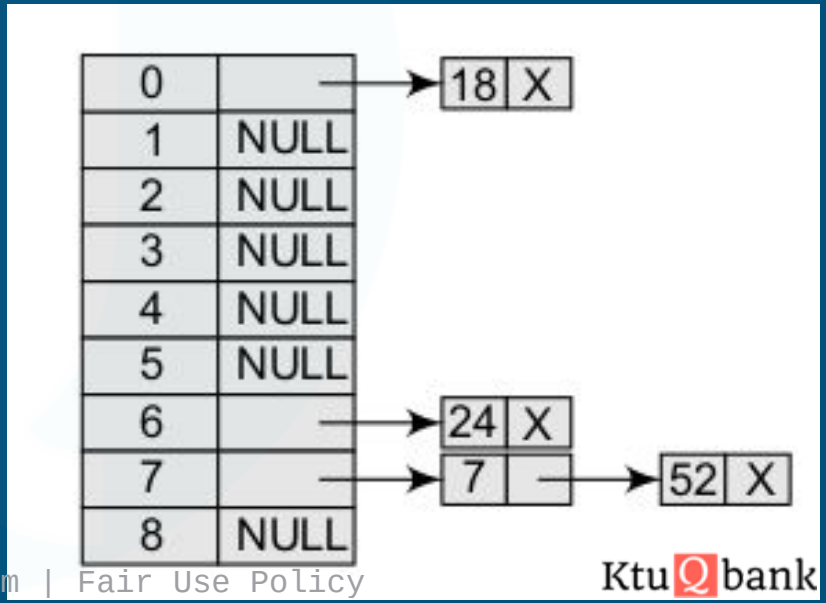
Create a linked list for location 0 and store the key value 18 in it as its only node.



Step 4: Key = 52

h(k) = 52 mod 9 = 7

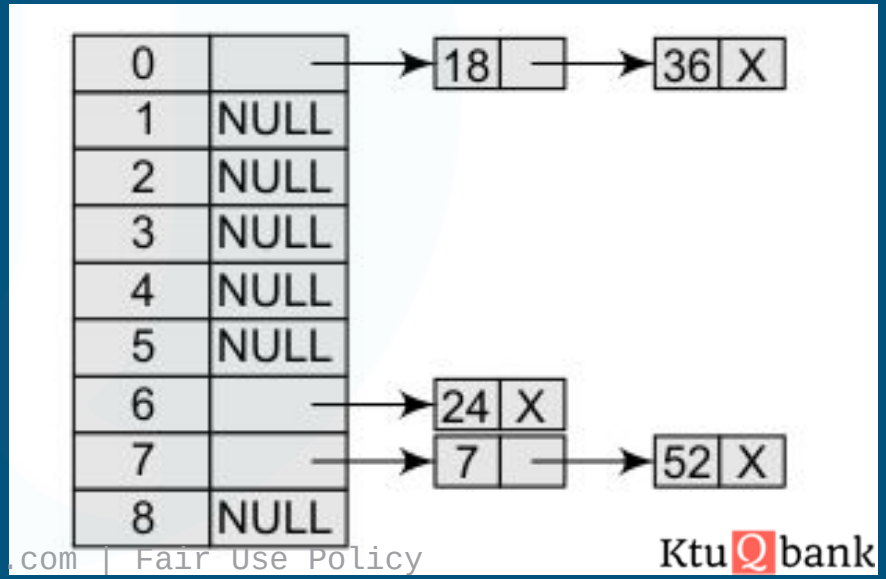
Insert 52 at the end of the linked list of location 7.



Step 5: Key = 36

h(k) = 36 mod 9 = 0

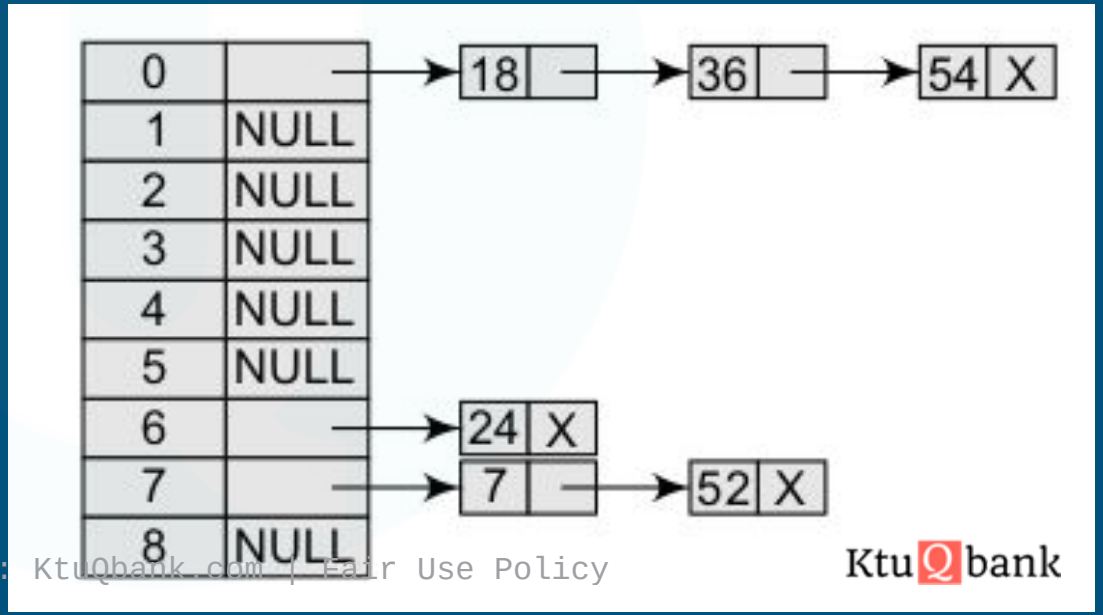
Insert 36 at the end of the linked list of location 0.



Step 6: Key = 54

h(k) = 54 mod 9 = 0

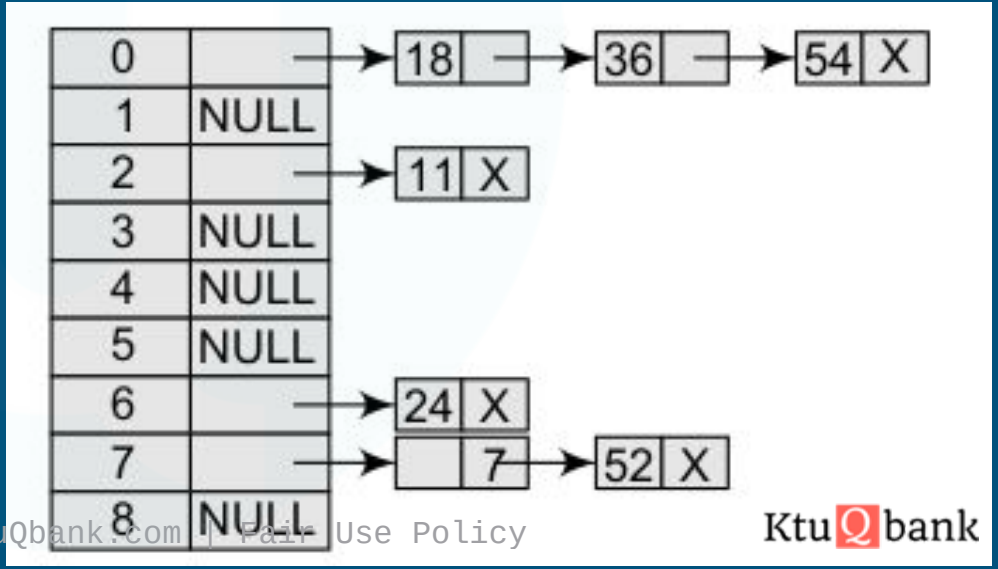
Insert 54 at the end of the linked list of location 0.



Step 7: Key = 11

h(k) = 11 mod 9 = 2

Create a linked list for location 2 and store the key value 11 in it as its only node.



Step 8: Key = 23

h(k) = 23 mod 9 = 5

Create a linked list for location 5 and store the key value 23 in it as its only node.