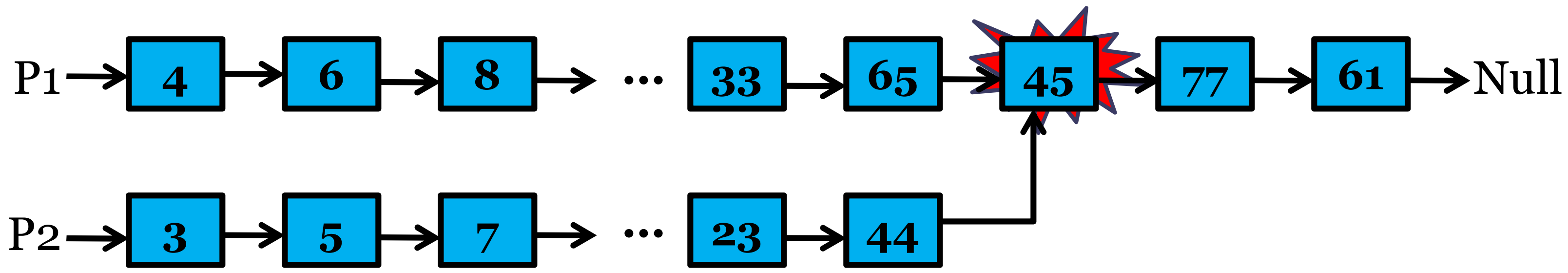
Review of Elementary Data Structures (Part 2)

Manoj Kumar
DTU, Delhi

Linked List: Problem

- Find the address/data of first common node .
- Use only constant amount of additional space.
- Your algorithm should run in O(m+n)



Hashing

Hashing : Introduction

- Suppose that we want to store 10,000 students records (each with a 5-digit ID) in a given container. Search operation on:
  - A linked list implementation would take **O(n)time.**
  - A height balanced tree would give **O(log n)access time.**
  - Using an array of size 100,000 would give **O(N) time, but will lead** to a lot of space wastage.
- Is there some way that we could get **O(1) search time without wasting a lot of space?**
- The answer is **hashing.**
  - Data stored using hashing is called **hashtable.**

DTU
Delhi Technological University

Hashing

- a data structure in which finds/searches are very fast
  - As close to O(1) as possible
  - minimum number of executed instructions per method
- Insertion and Deletion should be fast too
- Objects stored in hash table have unique keys
  - A key may be a single property/attribute value
  - Or may be created from multiple properties/values

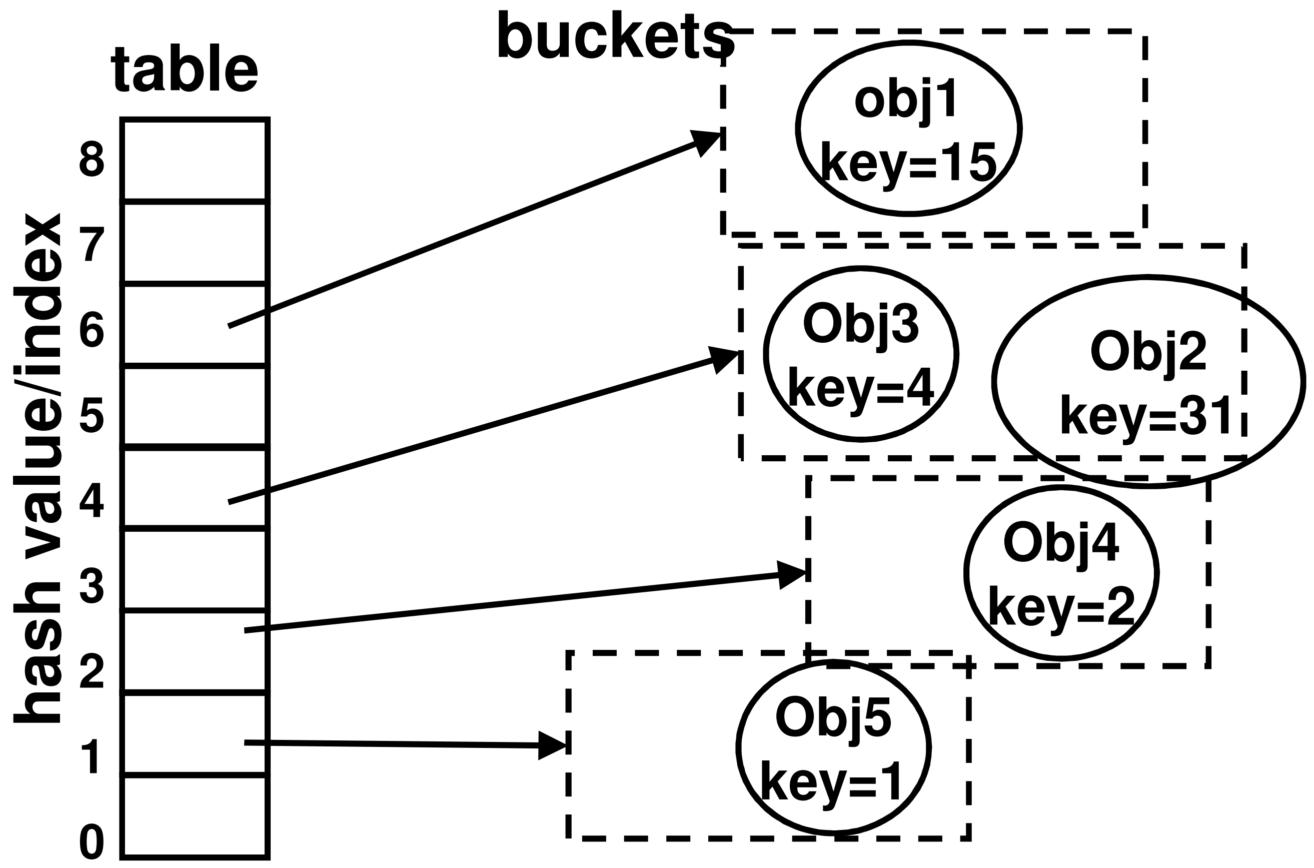
Hash tables vs. Other Data Structures

- We want to implement the operations Insert(), Delete() and Search()/Find() efficiently.
- Arrays:
  - can accomplish in O(1) time
  - but are not space efficient (assumes we leave empty space for keys not currently in dictionary)
- Binary search trees
  - can accomplish in O(log n) time
  - are space efficient.
- Hash Tables:
  - A generalization of an array that under some reasonable assumptions is O(1) for Insert/Delete/Search of a key

Hash Table

- Very useful data structure
  - Good for storing and retrieving key-value pairs
  - Not good for iterating through a list of items
- Example applications:
  - Storing objects according to ID numbers
    - When the ID numbers are widely spread out
    - When you don’t need to access items in ID order

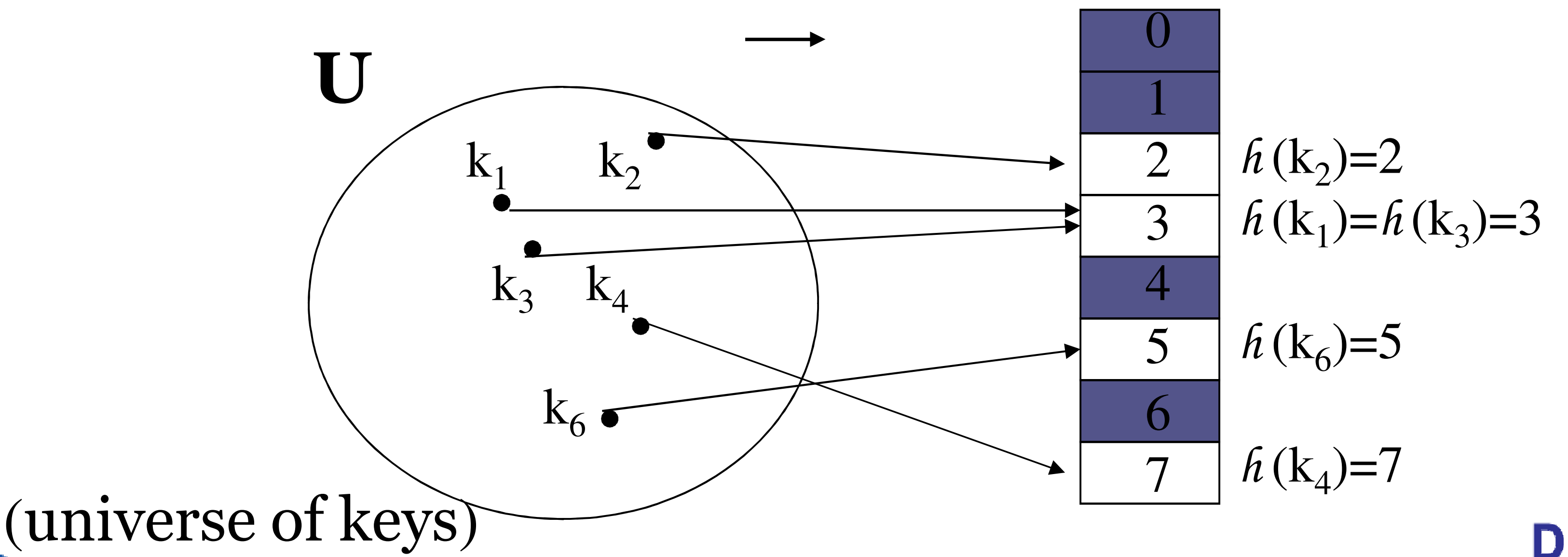
Hash Tables – Conceptual View



Hash Table

- **Hash Tables** solve these problems by using a much smaller array and mapping keys with a **hash function.**
- Let universe of keys U and an array of size m. A *hash function* h is a function from U to 0…m, that is:

$$h : U \longrightarrow 0 \ldots m$$



Hash index/value

- A *hash value* or *hash index* is used to index the hash table (array)
- A hash function takes a *key* and returns a hash value/index
  - The hash index is a integer (to index an array)
- The key is specific value associated with a specific object being stored in the hash table
  - It is important that the key remain constant for the lifetime of the object

Hash Function

- You want a hash function/algorithm that is:
  - Fast
  - Creates a good distribution of hash values so that the items (based on their keys) are distributed evenly through the array
- Hash functions can use as input
  - Integer key values
  - String key values
  - Multipart key values
    - Multipart fields, and/or
    - Multiple fields

The mod function

- Stands for **modulo**
- When you divide x by y, you get a result and a remainder
- Mod is the remainder
  - 8 mod 5 = 3
  - 9 mod 5 = 4
  - 10 mod 5 = 0
  - 15 mod 5 = 0
- Thus for key-value mod M, multiples of M give the same result, 0
  - But multiples of other numbers do not give the same result
  - So what happens when M is a prime number where the keys are not multiples of M?

Hash Tables: Insert Example

- For example, if we hash keys 0…1000 into a hash table with 5 entries and use ***h*(key) = key mod 5** , we get the following sequence of events:

Insert 2

	key	data
0		
1		
2	2	…
3		
4		

Insert 21

	key	data
0		
1	21	…
2	2	…
3		
4		

Insert 34

	key	data
0		
1	21	…
2	2	…
3		
4	34	…

Insert 54

There is a ***collision*** at array entry #4

???

What do we do about Collisions?

- Find a better hashing algorithm
  - Collisions occur when two or more records compete for the same address. Therefore we should try to find a hashing algorithm that distributes records fairly evenly among the available addresses
- Use a bigger table
  - The more free slots in the table, the less likely there will be a collision. But if you are doing lots of accesses, using a bigger table will reduce the likelihood that two accesses will reference the same part of the disk
- Need a system to deal with collisions

Dealing with Collisions

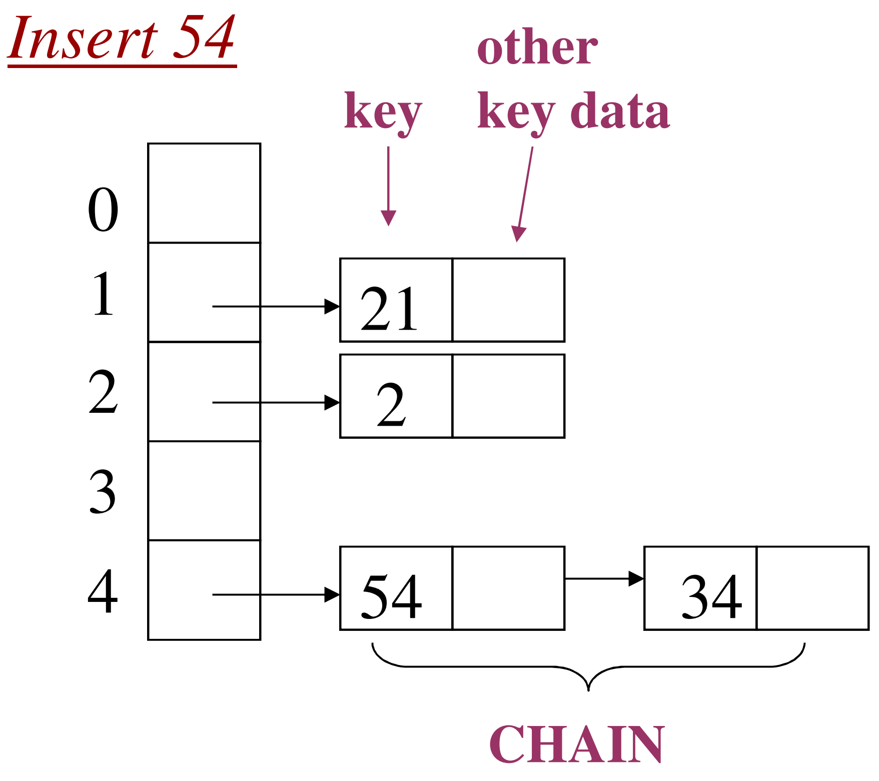
- A problem arises when we have two keys that hash in the same array entry – this is called a **collision**.
- There are three ways to resolve collision:
  - **Hashing with Buckets:** multiple records are stored in a bucket (block)
  - **Hashing with Chaining ("Separate Chaining"):** every hash table entry contains a pointer to a linked list of keys that hash in the same entry
  - **Hashing with Open Addressing:** every hash table entry contains only one key. If a new key hashes to a table entry which is filled, systematically examine other table entries until you find one empty entry to place the new key

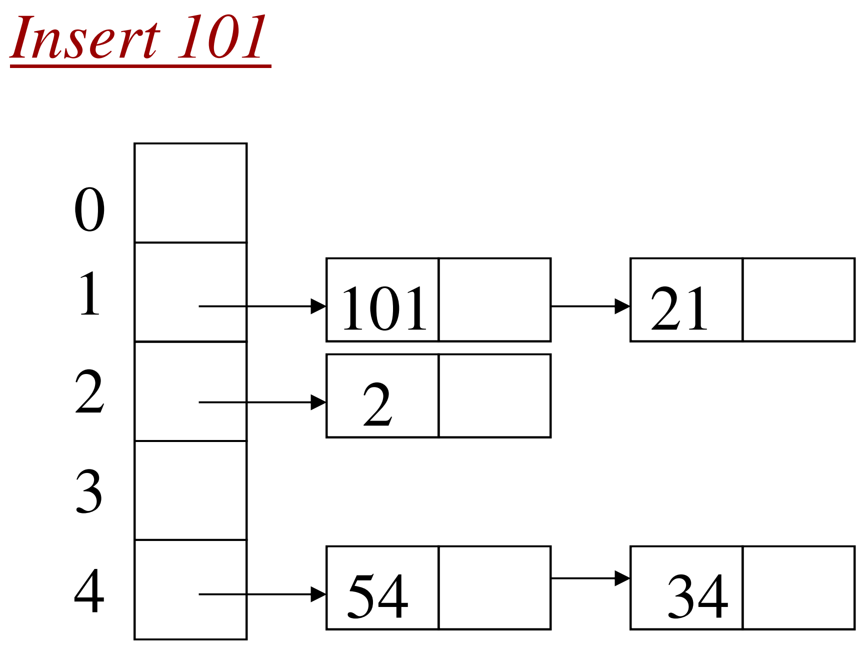
Hashing with buckets

- A common strategy is to have space for more than one record at each location in the hash table.
- Each location contains a bucket (or block or page) of records.
- Each bucket contains a fixed number of records, known as the blocking factor.
- The size of each bucket is set to the size of the block of data that is read in on each disk read.
- Reading in a whole bucket takes one disk access

Hashing with Chaining

- The problem is that keys 34 and 54 hash in the same entry (4). We solve this *collision* by placing all keys that hash in the same hash table entry in a **chain** (linked list) **or bucket** (array) pointed by this entry:





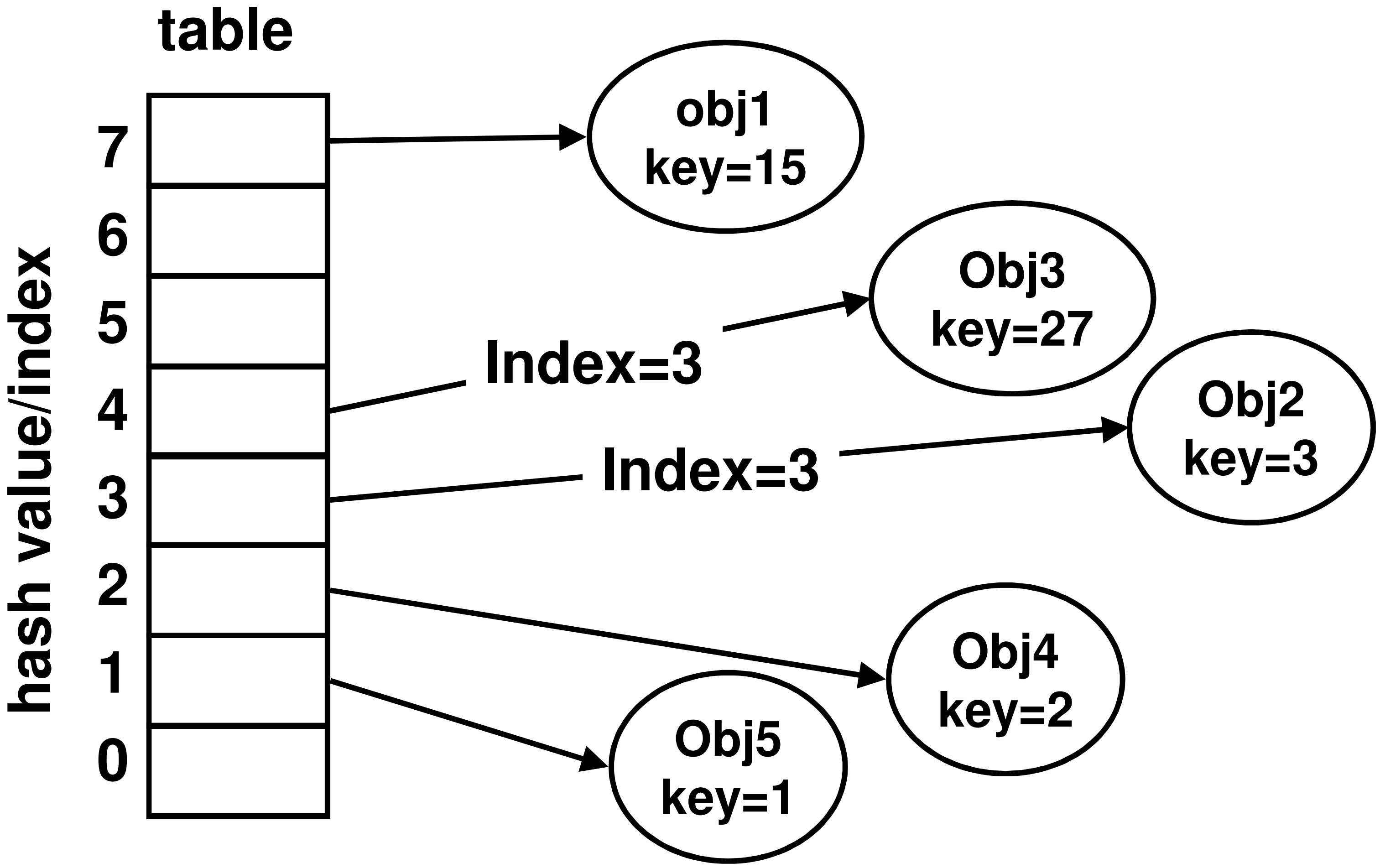
Hashing with Chaining

- What is the running time to insert/search/delete?
  - **Insert:** It takes O(1) time to compute the hash function and insert at head of linked list
  - **Search:** It is proportional to max linked list length
  - **Delete:** Same as search

- Therefore, in the unfortunate event that we have a "bad" hash function all *n* keys may hash in the same table entry giving an O(*n*) run-time!

So how can we create a "good" hash function?

Hash Tables – Open Addressing



Hashing with Open Addressing

- So far we have studies hashing with chaining, using a list to store the items that hash to the same location
- Another option is to store all the items (references to single items) directly in the table.
- **Open addressing**
  - collisions are resolved by systematically examining other table indexes, i_0 , i_1 , i_2 , … until an empty slot is located.

Open Addressing

- The key is first mapped to an array cell using the hash function (e.g. key % array-size)
- If there is a collision, find an available array cell
- There are different algorithms to find (to probe for) the next array cell
  - Linear
  - Quadratic
  - Double Hashing

Probe Algorithms (Collision Resolution)

- Linear Probing
  - Choose the next available array cell
    - First try arrayIndex = hash value + 1
    - Then try arrayIndex = hash value + 2
    - Be sure to wrap around the end of the array!
    - arrayIndex = (arrayIndex + 1) % arraySize
    - Stop when you have tried all possible array indices
  - If the array is full, you need to throw an exception or, better yet, resize the array

Probe Algorithms (Collision Resolution)

- Quadratic Probing
  - Variation of linear probing that uses a more complex function to calculate the next cell to try
    - First try arrayIndex = hash value + 1^2
    - Then try arrayIndex = hash value + 2^2
    - Be sure to wrap around the end of the array!
    - arrayIndex = (arrayIndex + i^2) % arraySize

Probe Algorithms (Collision Resolution)

Double Hashing

- Apply a second hash function after the first
- The second hash function, like the first, is dependent on the key
- Secondary hash function must
  - Be different than the first
- Good algorithm:
  - arrayIndex = (arrayIndex + stepSize) % arraySize;
  - Where stepSize = constant – (key % constant)
  - And constant is a prime less than the array size

Load factor

- Understanding the expected load factor will help you determine the efficiency of you hash table implementation and hash functions
- Load factor = number of items in hash table / array size
- For Open Addressing:
  - If < 0.5, wasting space
  - If > 0.8, overflows significant
- For Chaining:
  - If < 1.0, wasting space
  - If > 2.0, then search time to find a specific item may factor in significantly to the [relative] performance

Hashing functions

- Our goal in choosing any hashing algorithm is to spread out the records as uniformly as possible over the range of addresses available.
- Mod function
  - Let N be the maximum number of records expected.
  - Choose a prime number $p > N$
  - Hash function: $h(key) = key \bmod p$

Other Hash functions

Truncation or Digit/Character Extraction

- Work based on the distribution of digits or characters in the key.
- More evenly distributed digit positions are extracted and used for hashing purposes.
- For instance, students IDs or ISBN codes may contain common subsequences which may increase the likelihood of collision.
- Very fast, but digits/characters distribution in keys may not be very even.

Other Hash functions

Folding

- It involves splitting keys into two or more parts and then combining the parts to form the hash addresses.
- To map the key 25936715 to a range between 0 and 9999, we can:
  - split the number into two as 2593 and 6715 and
  - add these two to obtain 9308 as the hash value.
- Very useful if we have keys that are very large.
- Fast and simple especially with bit patterns.
- A great advantage is ability to transform non-integer keys into integer values.

Other Hash functions

Radix Conversion

- Transforms a key into another number base to obtain the hash value.
- Typically use number base other than base 10 and base 2 to calculate the hash addresses.
- To map the key 55354 in the range 0 to 9999 using base 11 we have:

  $\mathbf{55354_{10} = 38652_{11}}$

- We may truncate the high-order 3 to yield 8652 as our hash address within 0 to 9999.

Other Hash functions

Mid-Square

- The key is squared and the middle part of the result taken as the hash value.
- To map the key **3121 into a hash table of size 1000, we square it 3121^2 = 9740641 and extract 406 as the hash value.**
- Works well if the keys do not contain a lot of leading or trailing zeros.
- Non-integer keys have to be preprocessed to obtain corresponding integer values.

Some Applications of Hash Tables

- Database systems: Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.
- Symbol tables: The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.
- Data dictionaries: Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.
- Network processing algorithms: Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.

Sparse Matrices

$$\begin{pmatrix}
0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 11 & 0 & 0 \\
0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Sparse Matrices

- a matrix populated primarily with zeros.
- If most of the entries of a matrix are 0, then the matrix is said to be *sparse*

- In such a case, it may be very expensive to store zero values for two reasons:
  - The actual storage—if most entries are 0, do we need to store them?
  - Matrix operations—the results do not require computation:
    - $x + 0 = x$ and $x \cdot 0 = 0$

Dense and sparse matrix

	col 0	col 1	col 2	col 3	col 4
Row 0	-25	0	67	4	12
Row 1	1	8	45	61	2
Row 2	7	98	0	32	-5
Row 3	11	43	60	19	31

Dense matrix

	col 0	col 1	col 2	col 3	col 4	col 5	col 6
Row 0	16	0	0	0	12	0	0
Row 1	0	0	11	0	0	3	0
Row 2	0	5	0	0	0	0	6
Row 3	0	0	0	0	0	0	0
Row 4	13	0	0	0	0	0	4

Sparse Matrix

Sparse matrix: density

- The density of a matrix is defined as the ratio of non-zero entries over the total number of entries
  - A matrix with density around or less than 0.1 or 10% is usually considered to be sparse

Sparse Matrices

- For such sparse $N \times N$ matrices, we will
  - Denote the number of non-zero entries by m
- The *density* of a matrix is defined as the ratio of non-zero entries over the total number of entries
  - The density is $\dfrac{m}{N^2}$
  - The *row density* is the average number of non-zero values per row: $\dfrac{m}{N}$
  - A matrix with density less than 0.1 or $10\ \%$ is usually considered to be sparse
    - Usually, $m = \mathrm{O}(N)$

SAMSUNG

DTU Delhi Technological University

Sparse matrix representation

	col 0	col 1	col 2	col 3	col 4	col 5	Col 6
Row 0	16	0	0	0	12	0	0
Row 1	0	0	11	0	0	3	0
Row 2	0	5	0	0	0	0	6
Row 3	0	0	0	0	0	0	0
Row 4	13	0	0	0	0	0	4

General Matrix representation (Space required = 5x7x2 = 70 bytes)

#Rows

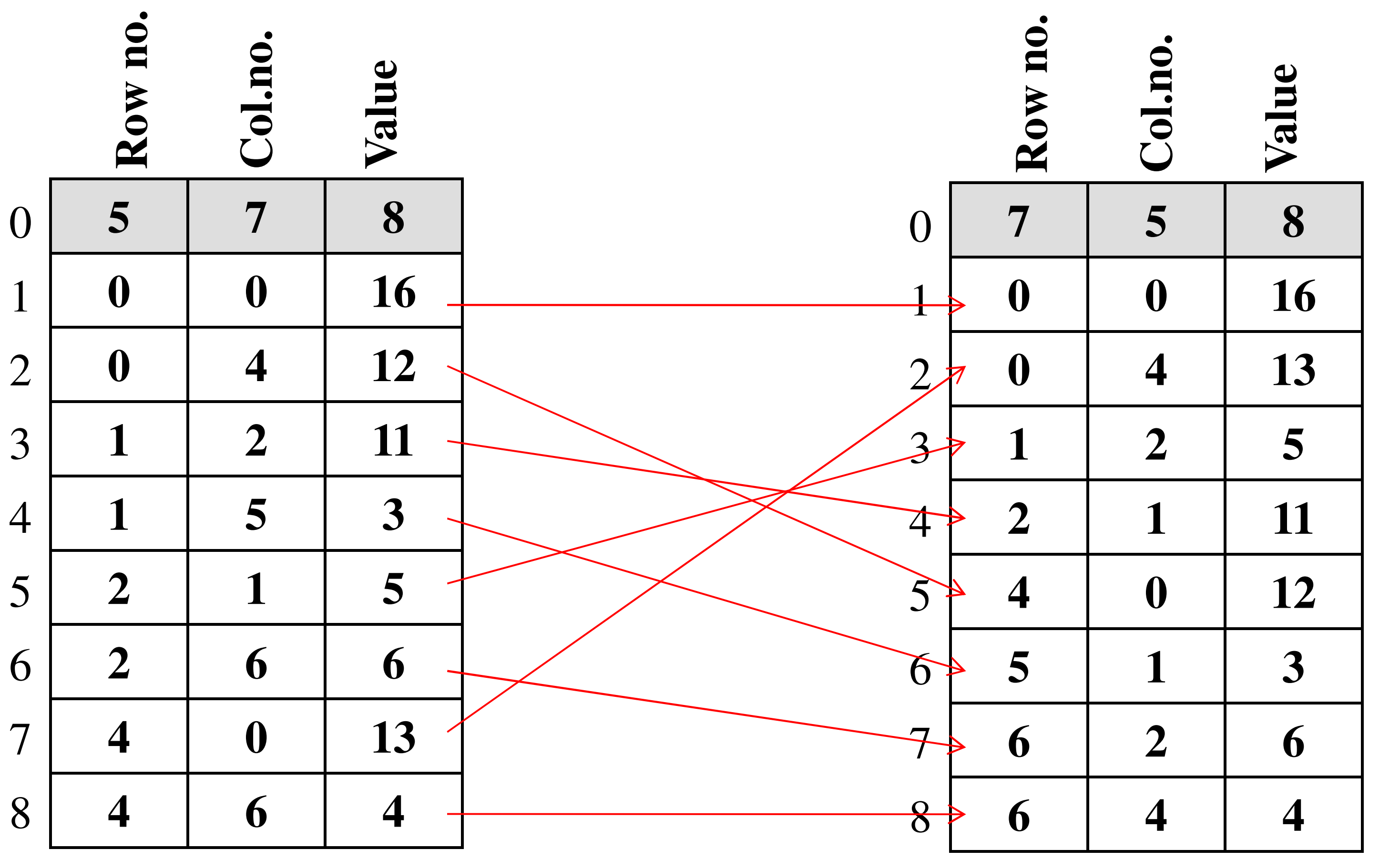
#Columns

#nonzero values

	Row no.	Col.no.	Value
0	5	7	8
1	0	0	16
2	0	4	12
3	1	2	11
4	1	5	3
5	2	1	5
6	2	6	6
7	4	0	13
8	4	6	4

Triplet representation (Space required = (1+8)x6 = 54 bytes)

Transpose of sparse matrix



	Row no.	Col.no.	Value
0	5	7	8
1	0	0	16
2	0	4	12
3	1	2	11
4	1	5	3
5	2	1	5
6	2	6	6
7	4	0	13
8	4	6	4

Sparse Matrix A

	Row no.	Col.no.	Value
0	7	5	8
1	0	0	16
2	0	4	13
3	1	2	5
4	2	1	11
5	4	0	12
6	5	1	3
7	6	2	6
8	6	4	4

Transpose of Matrix A

return *the matrix produced by interchanging the row and column value of every triple and then sort them by (row, column).*

Transpose: algorithm O(nt)

```
Transpose (sparse A[], sparse B[])
{
    m =A[0].row; n = A[0].col; t = A[0].value; // A is (m x n) matrix
    B[0].row = n;
    B[0].col = m;                                  // B is (n x m) matrix
    B[0].value = t;
    IF (t > 0) THEN
    {
            q = 1;
            FOR  cl = 0 TO n-1  DO                           //Transpose by columns
            {
                FOR p  = 1 TO  t   DO                        // Search for next element of column cl
                 {
                            IF (A[p].col = cl) THEN
                            {
                                        B[q].row = A[p].col;      //Copy from A to B
                                        B[q].col = A[p].row;
                                        B[q].value = A[p].value;
                                        q = q+1;
                            }
                 }
            }
    }       //end of IF
}
```

Fast Transpose: O(n+t)

- First determine number of elements in each column of matrix A.
- This gives us the number of elements in each row of transpose matrix B.
- Thus starting point of each row in matrix B can be easily computed.
- Now we can move elements from matrix A one by one into their correct position in B.

Fast Transpose algorithm

```
FastTranspose(Sparse A[], Sparse B[])
{
    n = A[0].col;   terms  = A[0].value;
    B[0].row = n;   B[0].col  = A[0].row;    B[0].value = terms;
    IF (terms > 0) THEN
    {
            FOR i = 0   TO n-1  DO   s[i] = 0;           // Compute s[i] = number of terms
            FOR i =1 TO terms   DO  s[A[i].col]++; // in row i of matrix B
            t[0] = 1;                             //  Compute t[i] = starting position of row i in B
            FOR i =1 TO n-1  DO t[i] = t[i-1] + s[i-1];
            FOR i = 1 TO terms DO                        // Move elements from A to B
            {
                    j = t[A[i].col];
                    B[j].row = A[i].col;
                    B[j].col  = A[i].row;
                    B[j].value = A[i].value;
                    t[A[i].col]  = j+1;
            }
    }
}
```

Addition of sparse matrices

	Row	Col	Value
0	5	7	6
1	0	0	9
2	0	4	7
3	2	3	11
4	3	5	13
5	4	1	5
6	4	6	8

Sparse Matrix A

+

	Row	Col	Value
0	5	7	5
1	0	4	5
2	1	2	14
3	3	5	4
4	3	6	8
5	4	6	3

Sparse Matrix B

=

	Row	Col	Value
0	5	7	8
1	0	0	9
2	0	4	12
3	1	2	14
4	2	3	11
5	3	5	17
6	3	6	8
7	4	1	5
8	4	6	11

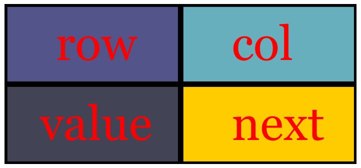
Sparse Matrix C

Sparse Matrix: Linked List representation

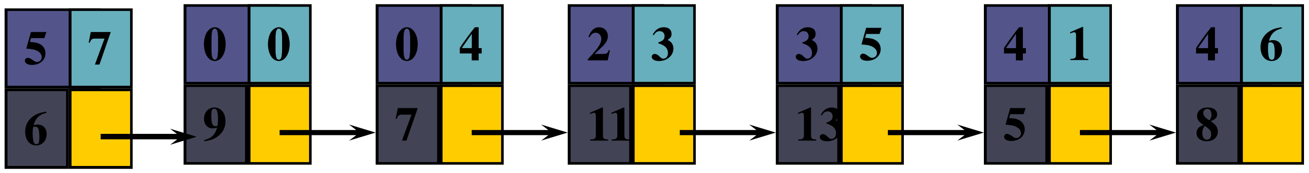
1. Single chain

0	**5**	**7**	**6**
1	**0**	**0**	**9**
2	**0**	**4**	**7**
3	**2**	**3**	**11**
4	**3**	**5**	**13**
5	**4**	**1**	**5**
6	**4**	**6**	**8**

Sparse Matrix A



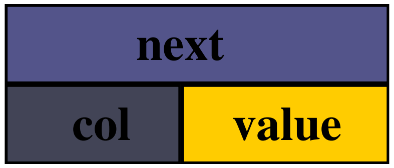
Node Structure



Head node

Sparse Matrix: Linked List representation

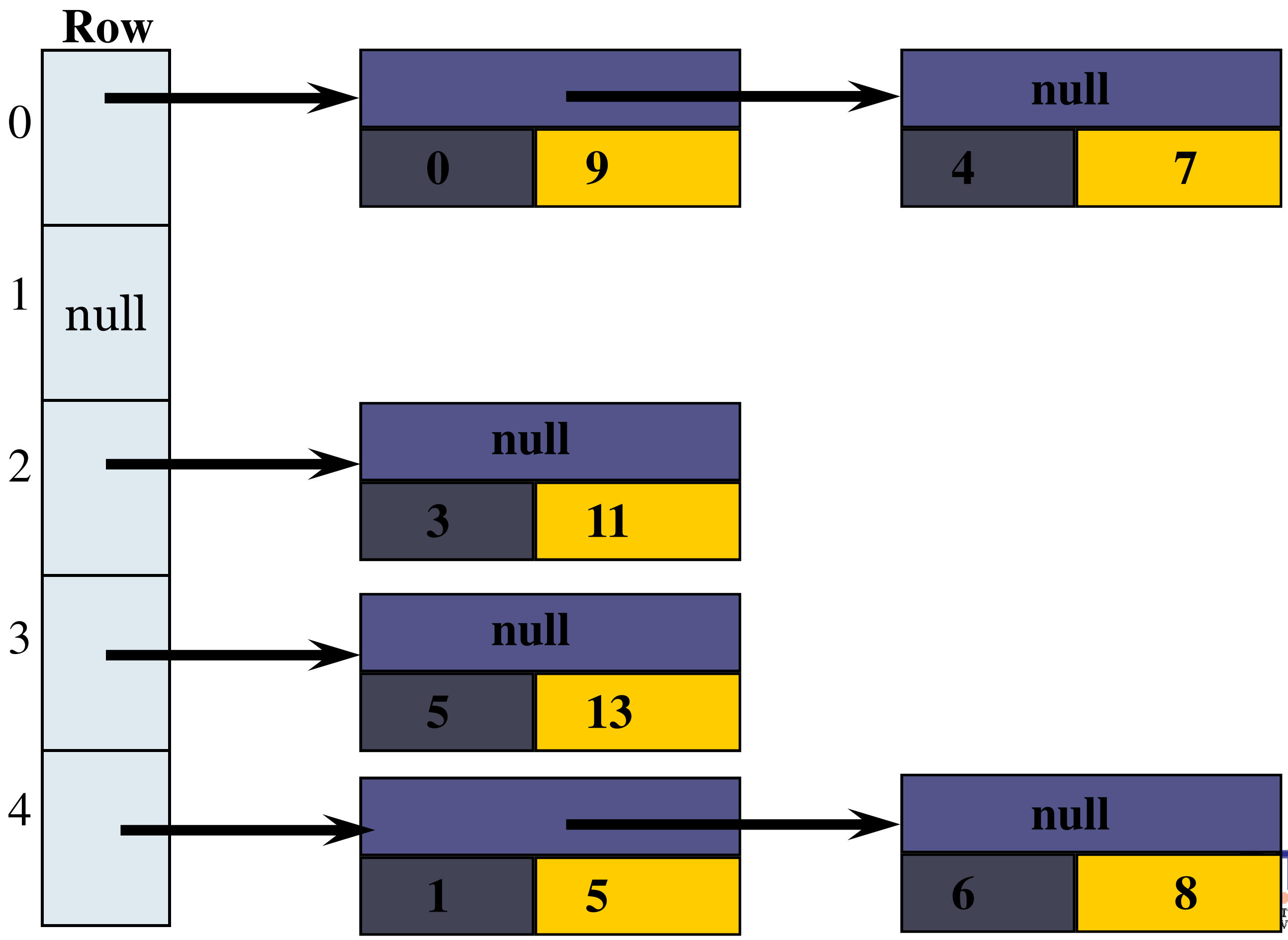
2. One Linked List Per Row



Node structure

0	5	7	6
1	0	0	9
2	0	4	7
3	2	3	11
4	3	5	13
5	4	1	5
6	4	6	8

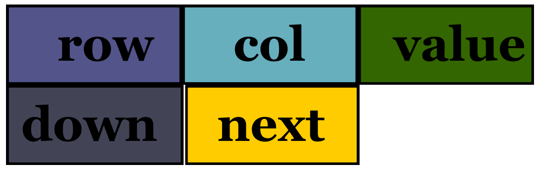
Sparse Matrix A



Sparse Matrix: Linked List representation

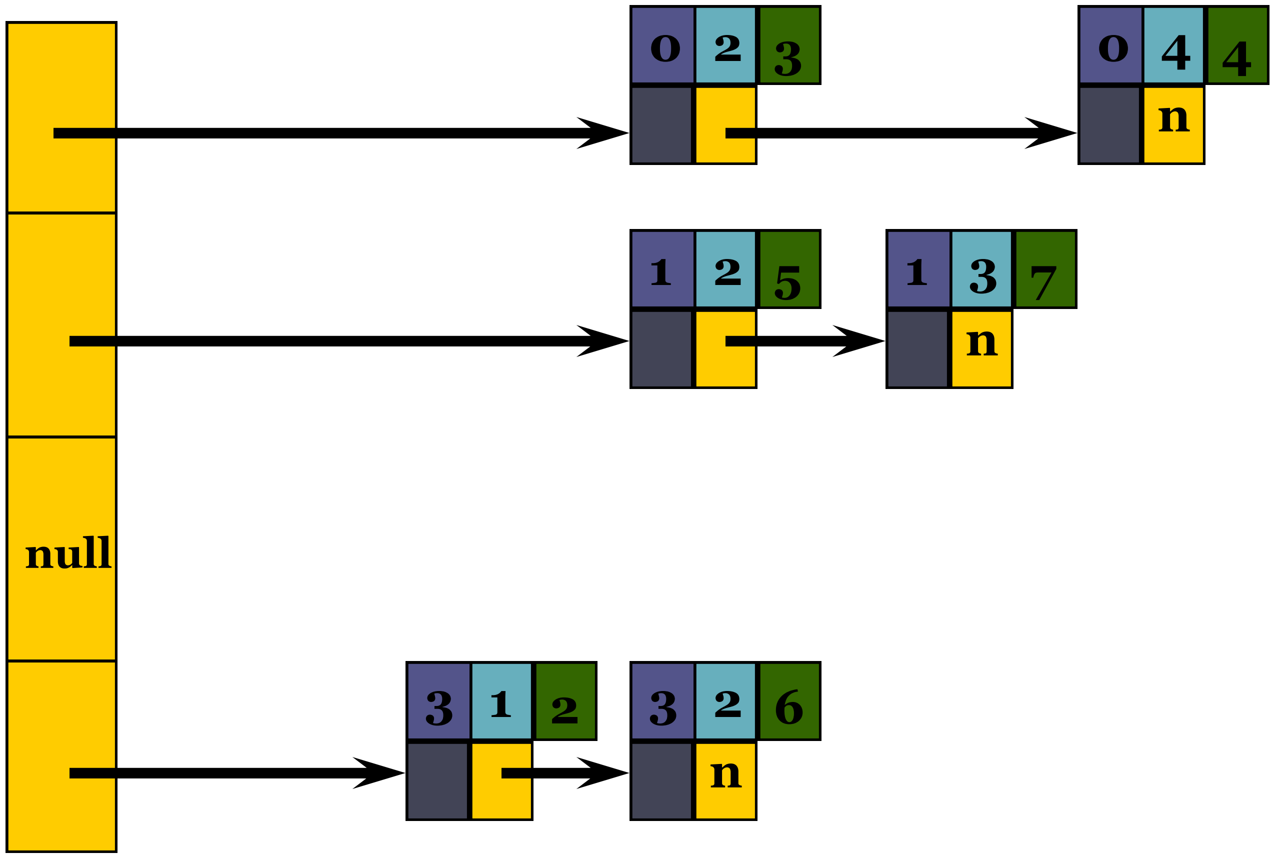
3. Orthogonal List Representation

Node structure



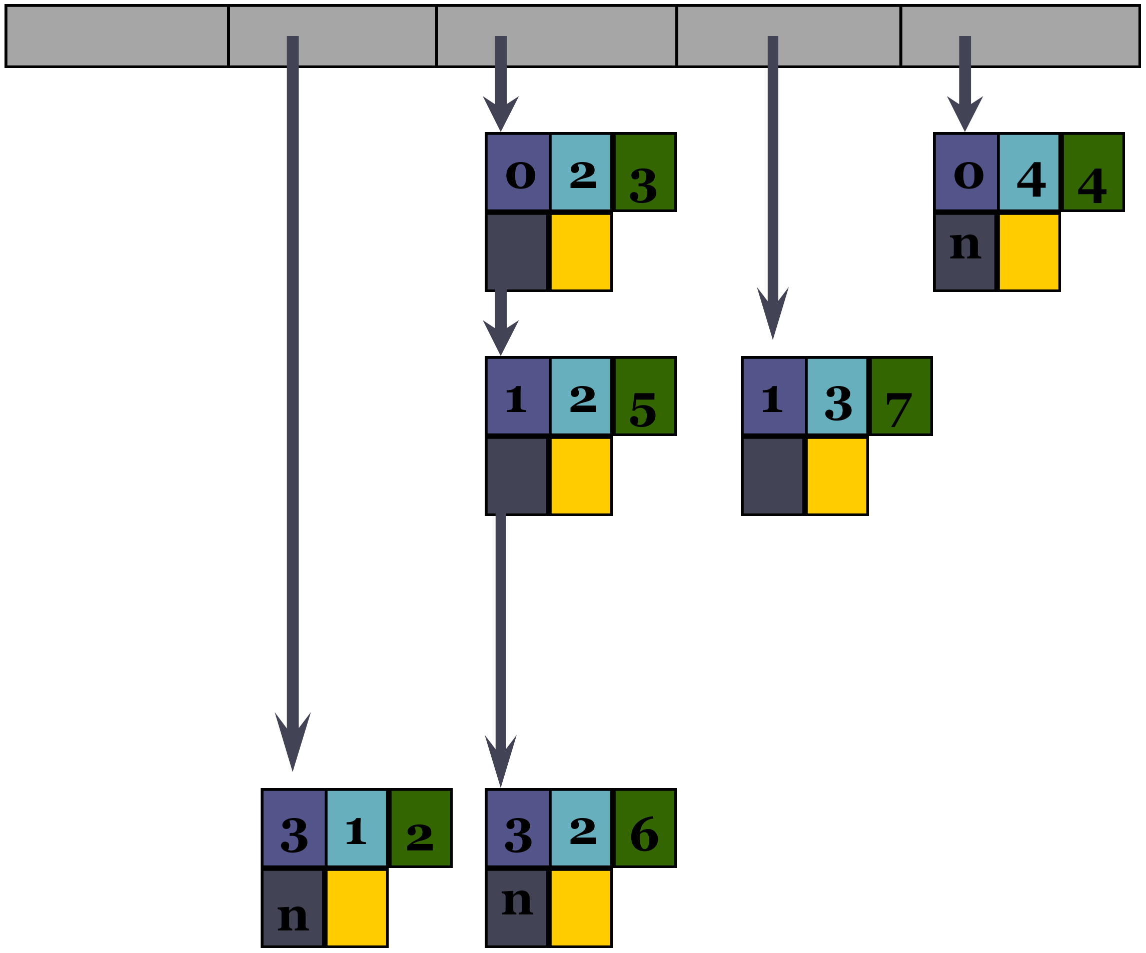
Row Lists

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0



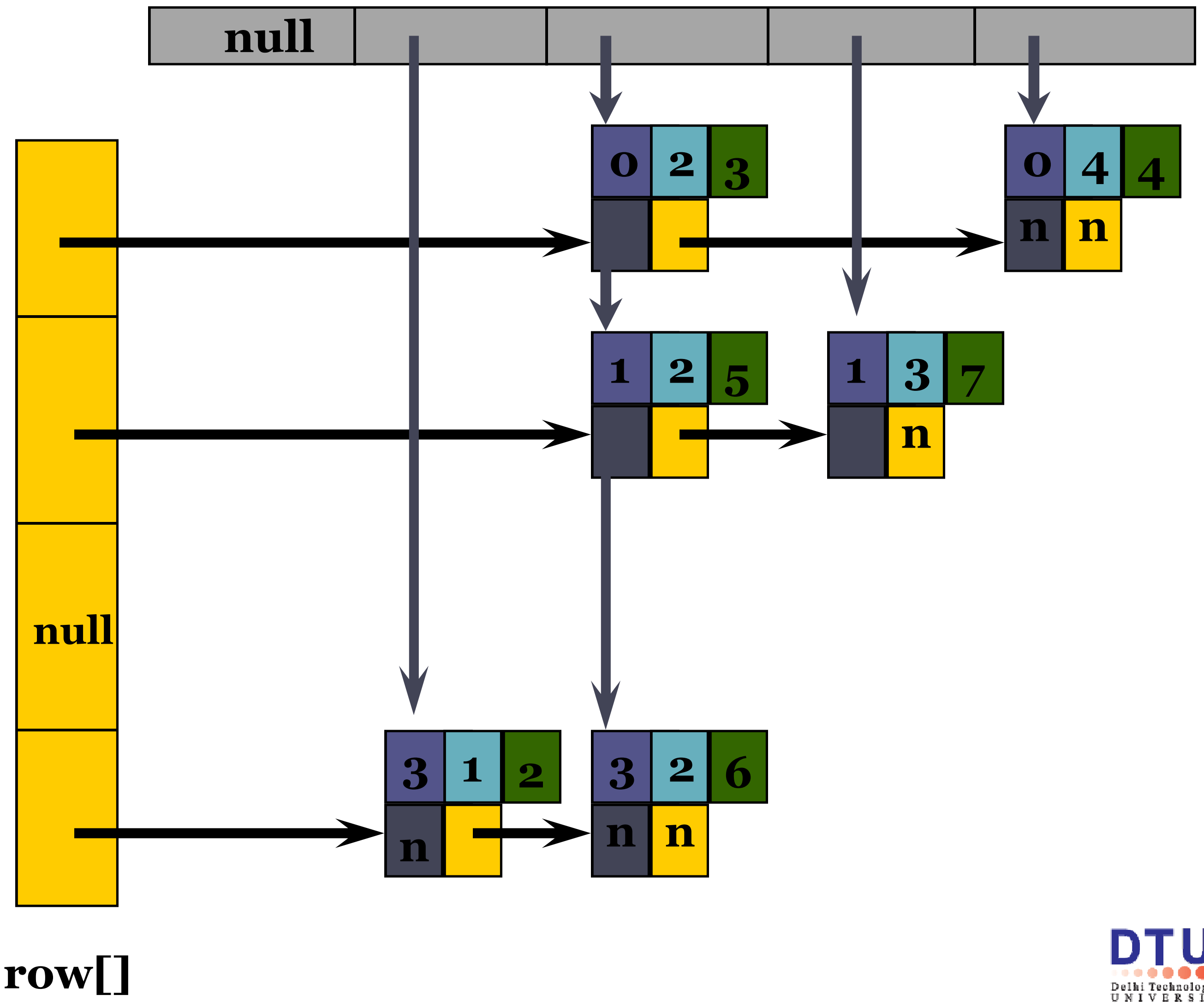
Column list

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0



Orthogonal List

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

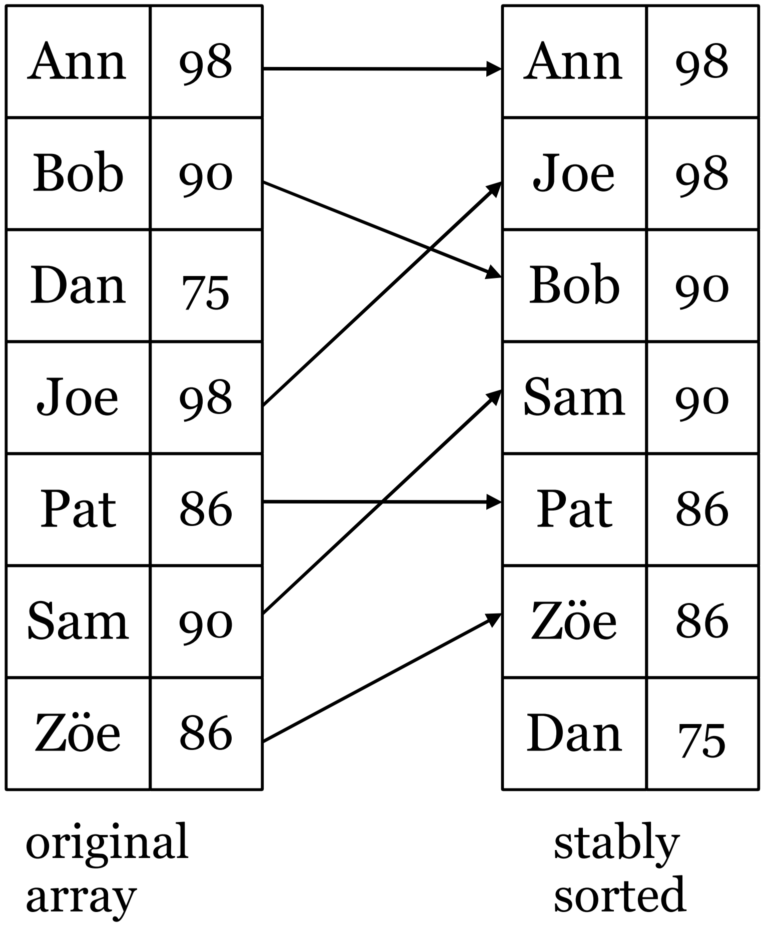


Sorting algorithms

- Selection sort
- Bubble sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort

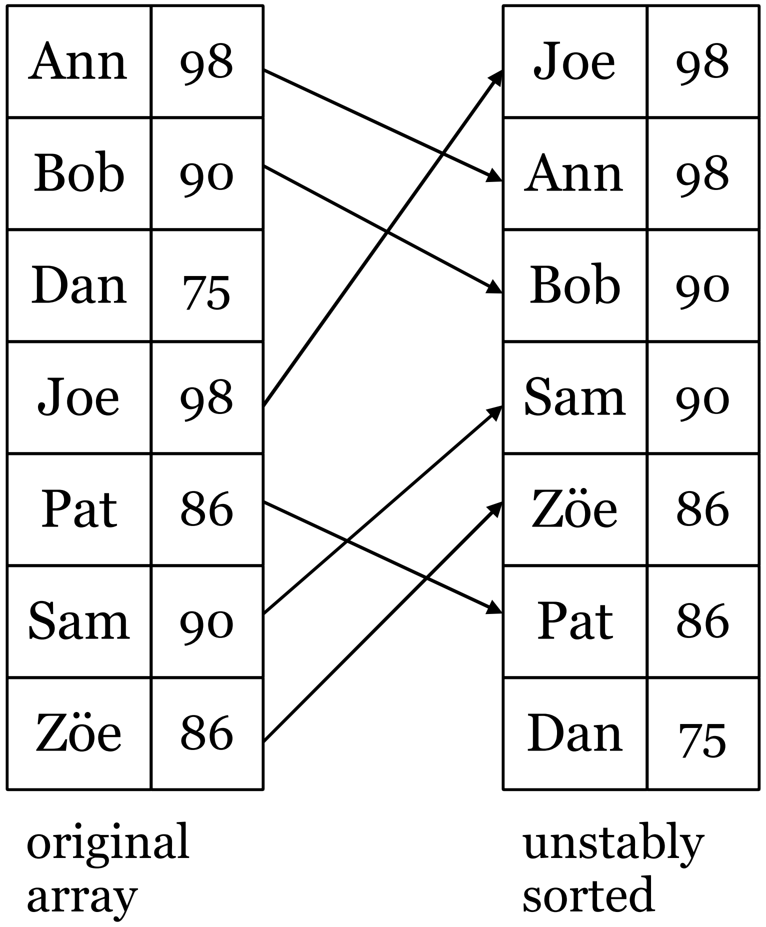
Stable sort algorithms

- A stable sort keeps equal elements in the same order
- This may matter when you are sorting data according to some characteristic
- Example: sorting students by test scores



Unstable sort algorithms

- An unstable sort may or may not keep equal elements in the same order
- Stability is usually not important, but sometimes it is important



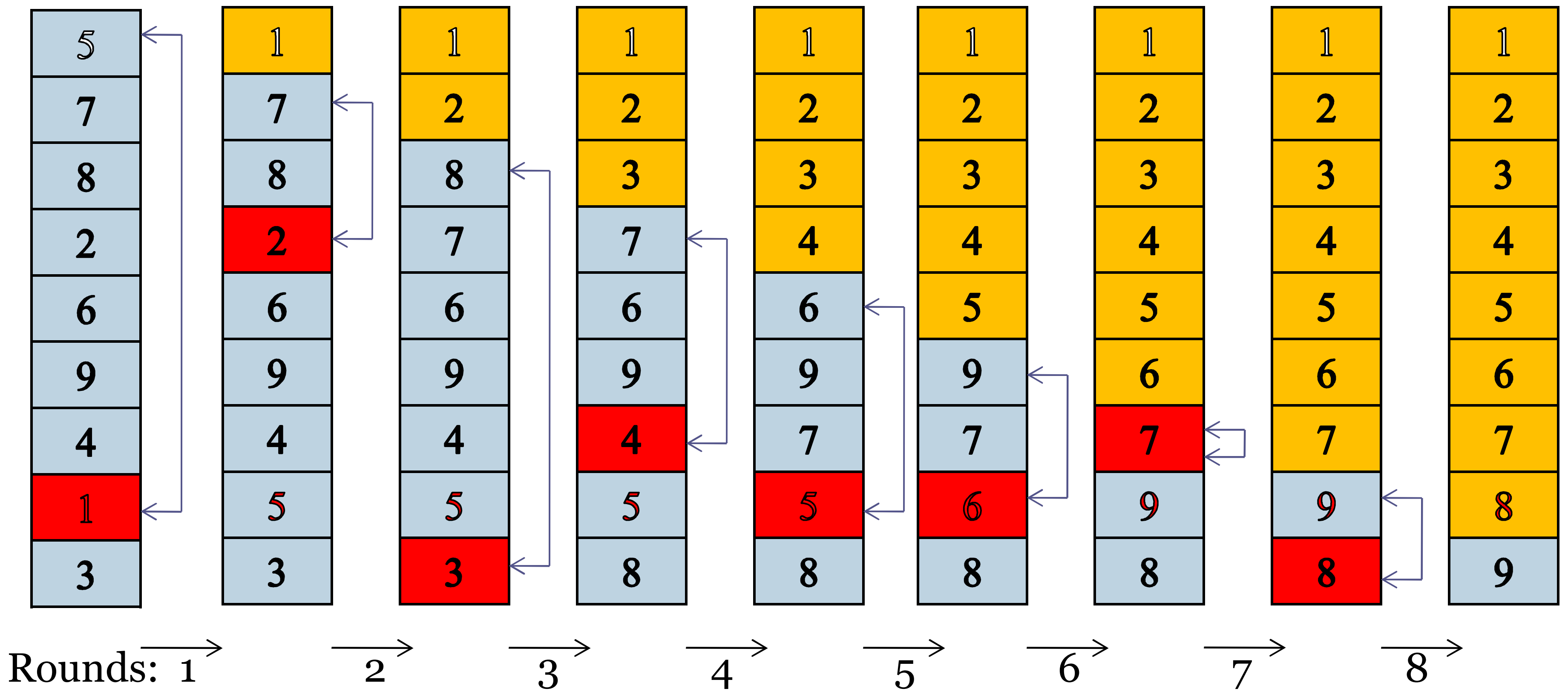
Selection Sort

The algorithm works as follows:

- Find the minimum value in the list
- Swap it with the value in the first position
- Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

- Effectively, the list is divided into two parts:
  - the sublist of items already sorted, which is built up from left to right and is found at the beginning, and
  - the sublist of items remaining to be sorted, occupying the remainder of the array.

Selection Sort



Total rounds: N-1

Round1: select smallest from N elements and exchange with 1st element.

Round 2: select smallest from N-1 elements and exchange with 2nd element

...

Round N-1: select smallest from 2 elements and exchange with (N-1)th element.

Selection Sort: Algorothm

```
Selection_Sort( A[], N)
{
   FOR i = 0 TO N-2 DO       // total N-1 rounds
   {
         minIndex = i;  // index of smallest
         FOR j = i+1 TO N-1 DO
         {
                   IF ( A[j] < A[minIndex]) THEN
                             minIndex=j;
         }
         swap  A[i]  ←→A[minIndex]);  //swap smallest with top
                                       //of array
   }
}
```

Selection Sort: Analysis

Total work done =

N+

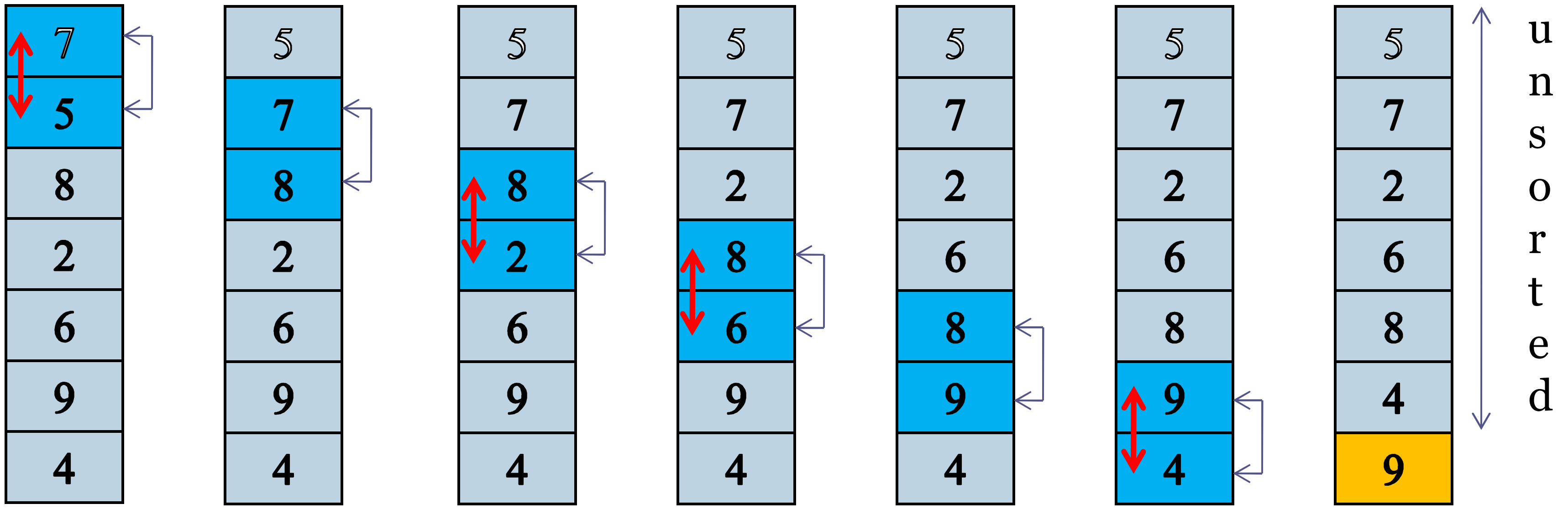
(N-1) +

(N-2) +

…

+ 2 $\approx N*(N-1)/2 = O(N^2)$

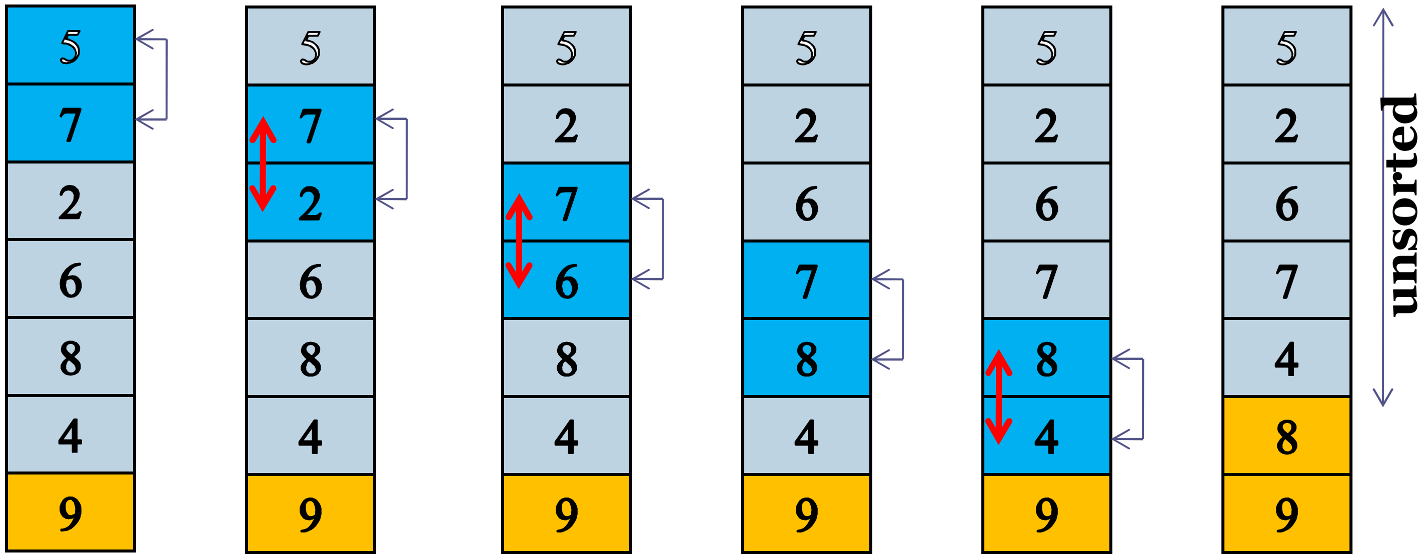
Bubble Sort



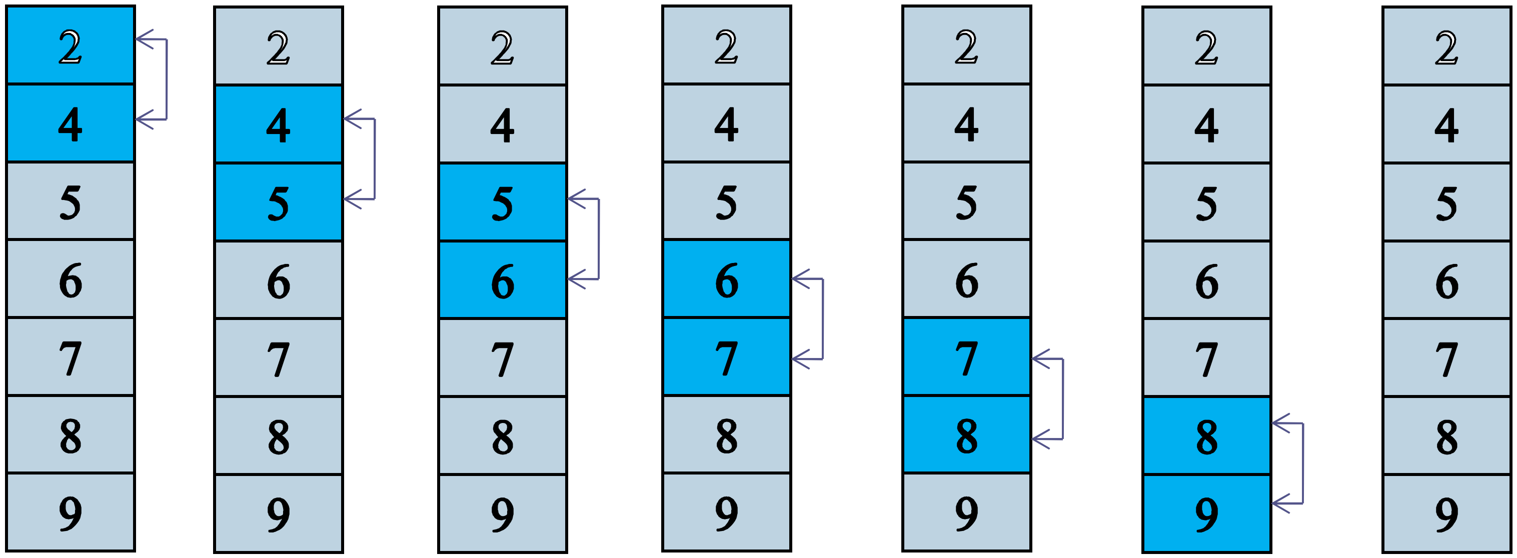
One round of Bubble Sort

Number of comparisons: N-1 (N-1 pairs)
Result of this round: largest element is settled at bottom
Next Round: repeat for first N-1 elements

Bubble Sort …



Bubble Sort: list is already sorted



No swapping performed during first round
Do not perform remaining round(s) is array is already sorted

Bubble Sort: Algorithm

```
Bubble_Sort(A[], N)
{
    FOR i = 1 TO N-1 DO                  // total N-1 rounds required
    {
            flag = 0;                    // to check if any element is swapped
            FOR j = 0 TO N-1-i DO
            {
                    IF (A[j] >A[j+1]) THEN
                    {
                            swap A[j] <-> A[j+1];
                            flag = 1;  //swapping done; set flag
                    }
            }
            IF (flag == 0) THEN  break;  // array is already sorted; skip remaining rounds
    }
}
```

Bubble Sort: Analysis

- Number of comparisons in round 1: N-1
- Number of comparisons in round 2: N-2

…..

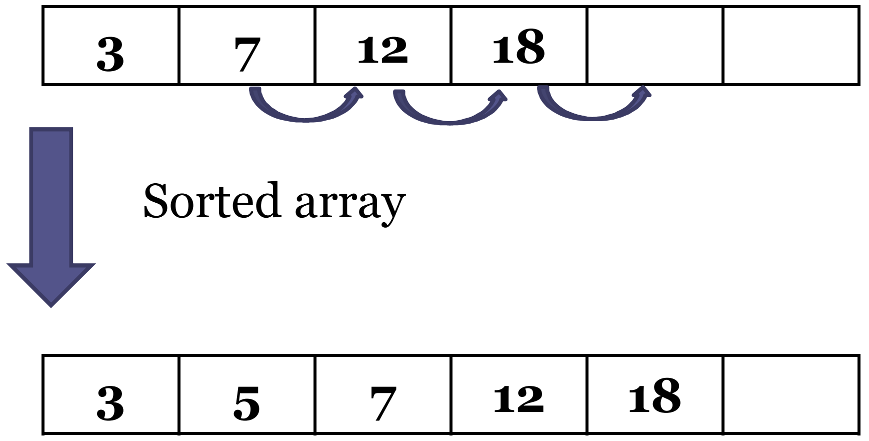
- Number of comparisons in round N-1: 1

- Total comparisons done : (N-1)+(N-2)+…. + 1

  = (N-1)*(N-2)/2

  = $O(N^2)$
- Best case: (array already sorted): Ω(N)

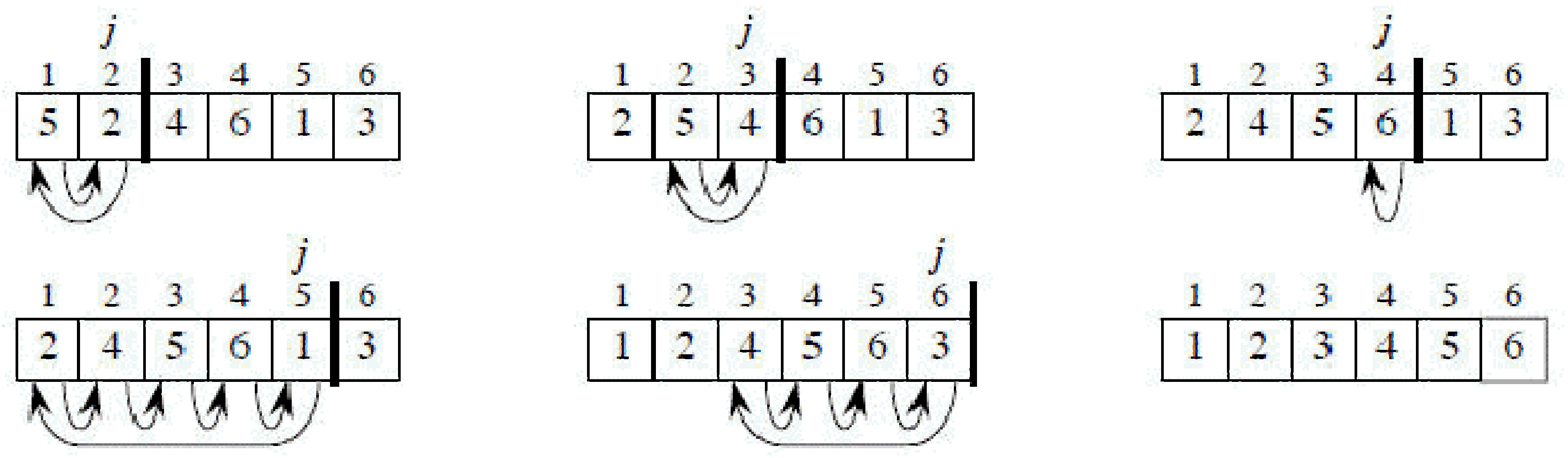
Insertion Sort



Key=5 to
be inserted

Insertion Sort

- Array containing only first element is sorted.
- Insert remaining elements one by one into sorted array.



Insertion Sort: Algorithm

```
Insertion_Sort(A[], N)
{
  FOR j = 1 TO N-1 DO
   {
        key = $A[j]$;
        //put $A[j]$ into the sorted sequence $A[0 . . j − 1]$
        $i = j − 1$;
        WHILE ($i > 0$ AND $A[i] >$ key) DO
        {
              $A[i +1] = A[i]$;
              $i = i − 1$;
        }
        $A[i + 1] =$ key;
   }
}
```

Insertion Sort

- Worst case Analysis
  - Initial sorted array size = 1
  - Shifting required to insert A[1] =1
  - Shifting required to insert A[2] =2
  - Shifting required to insert A[3] =3

……

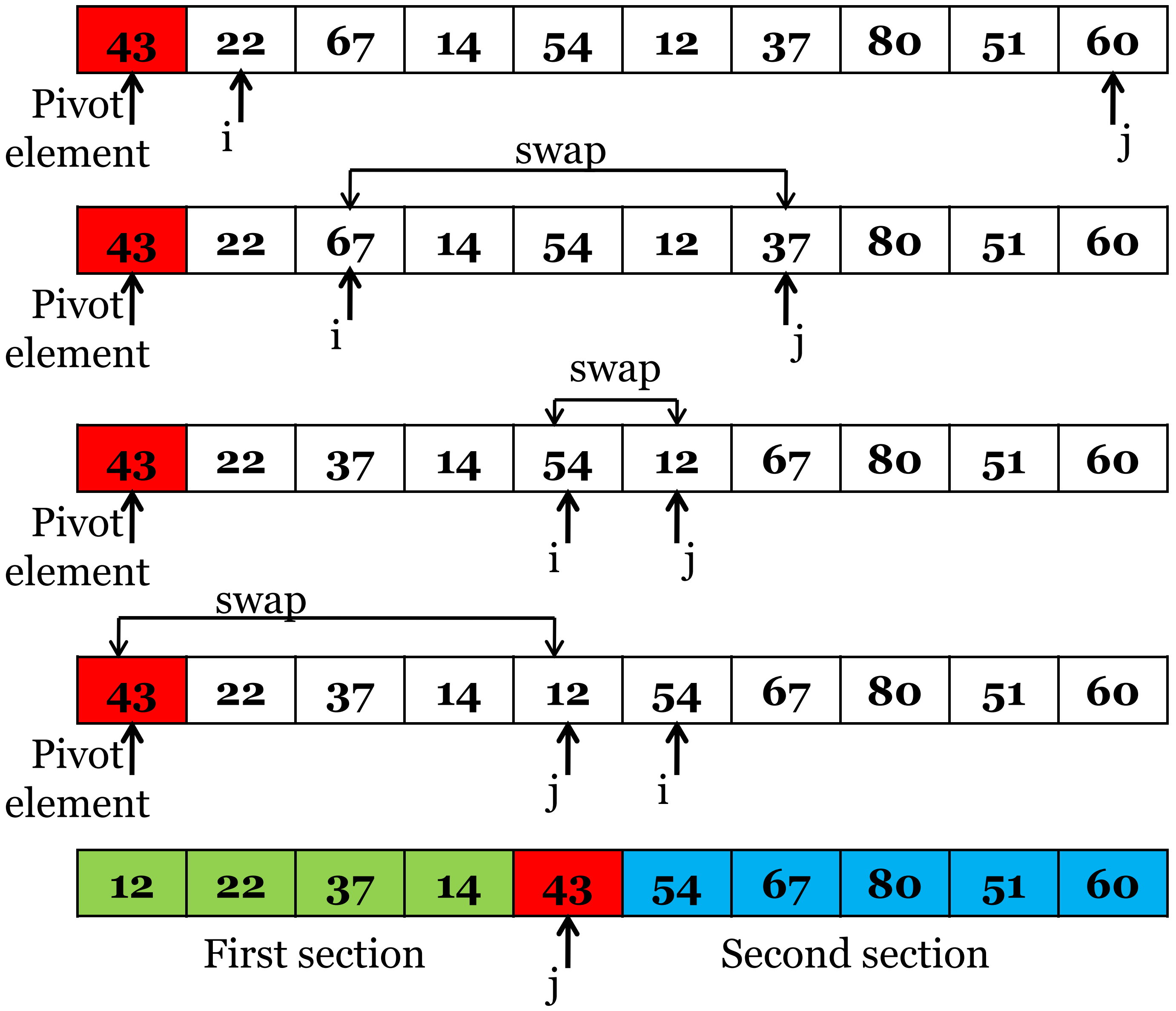
  - Shifting required to insert A[N-1] =N-1
- Total numbers shifted = 1+2+3+….+(N-1)

  $= O(N^2)$

Quick Sort

- Aim in each round:
  - Select a pivot element x (say first element)
  - Find correct position of x in array.
  - While doing this, move all numbers smaller than x, before x and all elements larger than x, after x.
  - Now array is divided into two parts
    - First: section of array before x (numbers < x)
    - Second: section of array after x (numbers > x)
  - Apply quick sort on these two sections.

Quick Sort : One pass



Quick_Sort: Algorithm

```
Quick_Sort(A[], first, last)
{
    IF  (first < last)  THEN
    {
            pivot = A[first];
            i = first;
            j = last;
            WHILE  (i  <  j) DO
            {
                    WHILE (A[i] <= A[pivot]  AND   i < last)  DO   i = i + 1;
                    WHILE ( A[j] > A[pivot] )  DO  j  = j - 1;
                    IF  (i < j)  THEN
                    {               // swap A[i] and A[j]
                                    swap   A[i] <-> A[j];
                    }
            }
            temp = A[pivot];
            A[pivot] = A[j];
            A[j] = temp;
            Quick_Sort (A, first, j-1);
            Quick_Sort (A, j+1, last);
    }
}
```

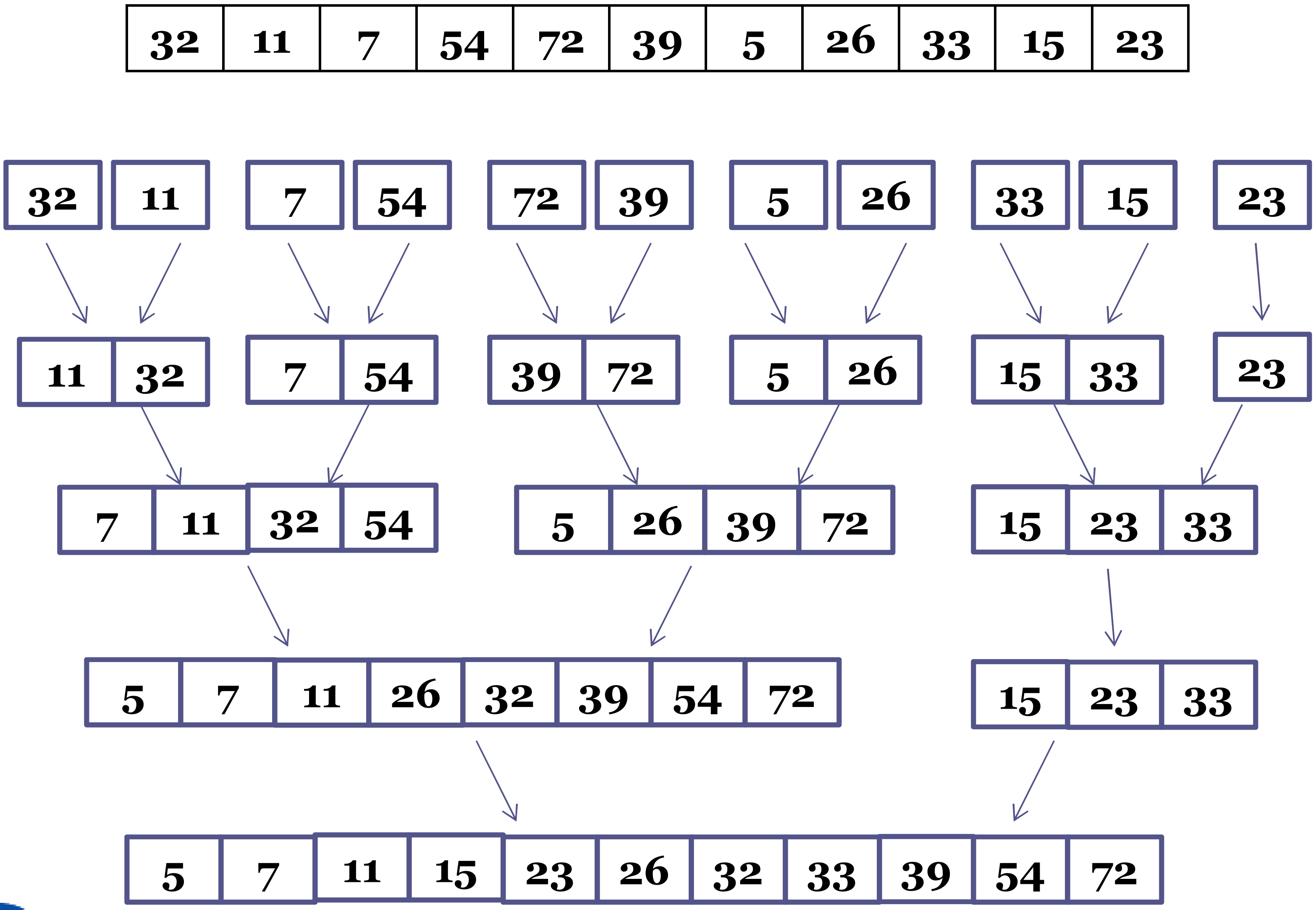
SAMSUNG

DTU Delhi Technological University

Quick Sort : Analysis

- Depth of recursion: log N times
- In every pass, max N elements are processed.
- Complexity: θ(N logN) average case
- Worst case: $O(N^2)$

Merge Sort


32 11 7 54 72 39 5 26 33 15 23
32
11
7
54
72
39
5
26
33
15
23
11 32
7 54
39 72
5 26
15 33
23
7 11 32 54
5 26 39 72
15 23 33
5 7 11 26 32 39 54 72
15 23 33
5 7 11 15 23 26 32 33 39 54 72

Merge Sort : a recursive algorithm

```
mergeSort(A[],left,right)
{       // sort A[left .. right]
  IF (left < right) THEN
  {    // at least two elements
    mid = (left+right)/2; //midpoint
    mergeSort(A, left, mid); //sort first half
    mergeSort(A, mid + 1, right); //sort other half
    merge(A, B, left, mid, right);    //merge from A to B
    copy(B, A, left, right);        //copy result back to A
  }
}
```

Merge Sort: Analysis

- Number of passes: Log(N)
- Number of elements processes in each pass: N
- Complexity: O(N logN)

Heap

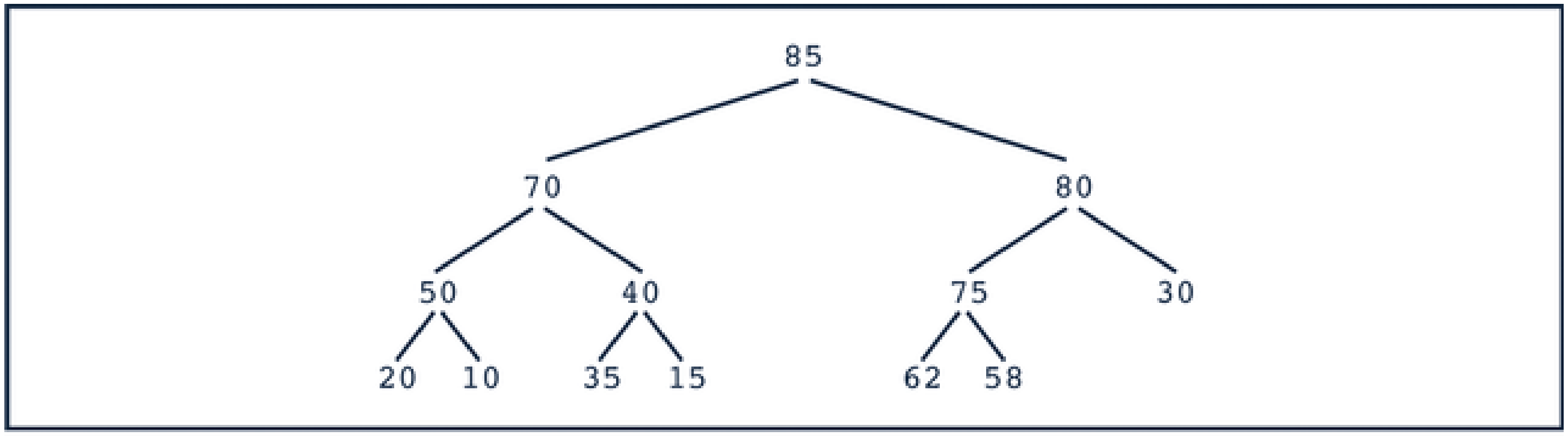
Definition:

A **heap** is a list in which each element contains a key, such that the key in the element at position k in the list is at least as large as the key in the element at position $2k + 1$ (if it exists), and $2k + 2$ (if it exists)

Heap

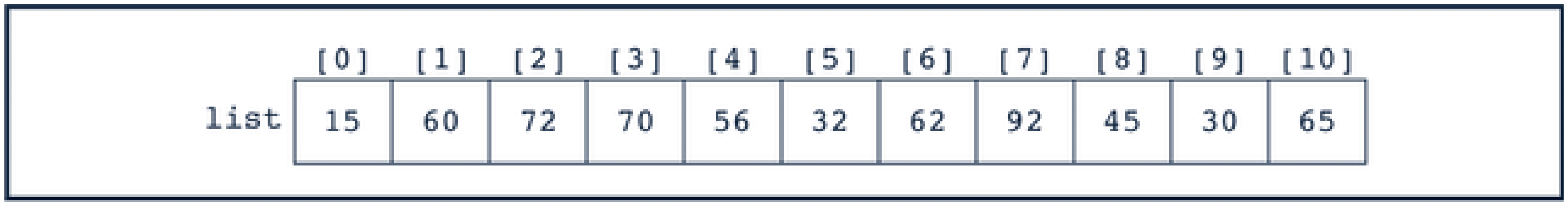
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
85	70	80	50	40	75	30	20	10	35	15	62	58

A list that is a heap



Complete binary tree corresponding to the list

Building a Heap



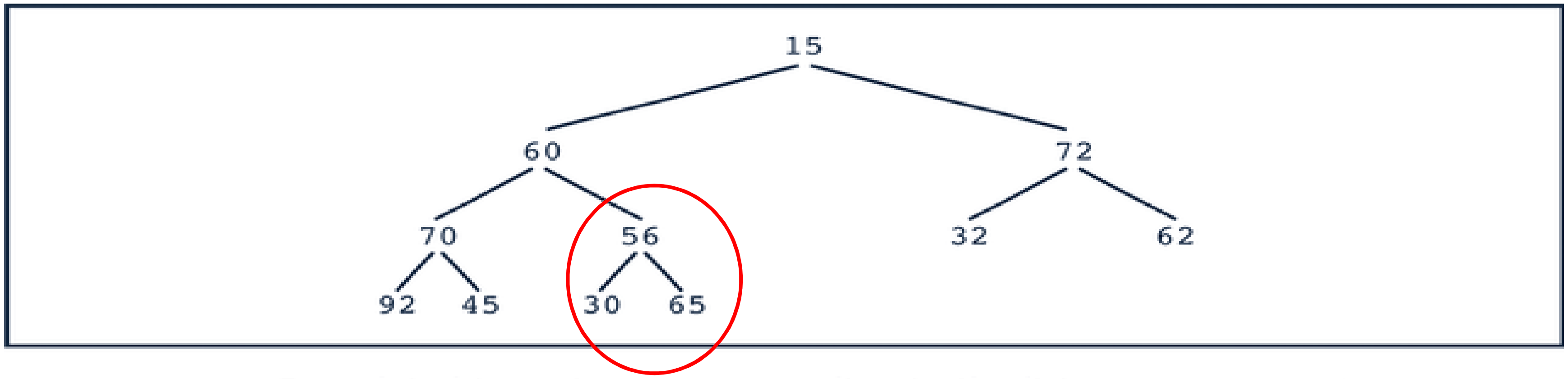
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
list	15	60	72	70	56	32	62	92	45	30	65

Array list

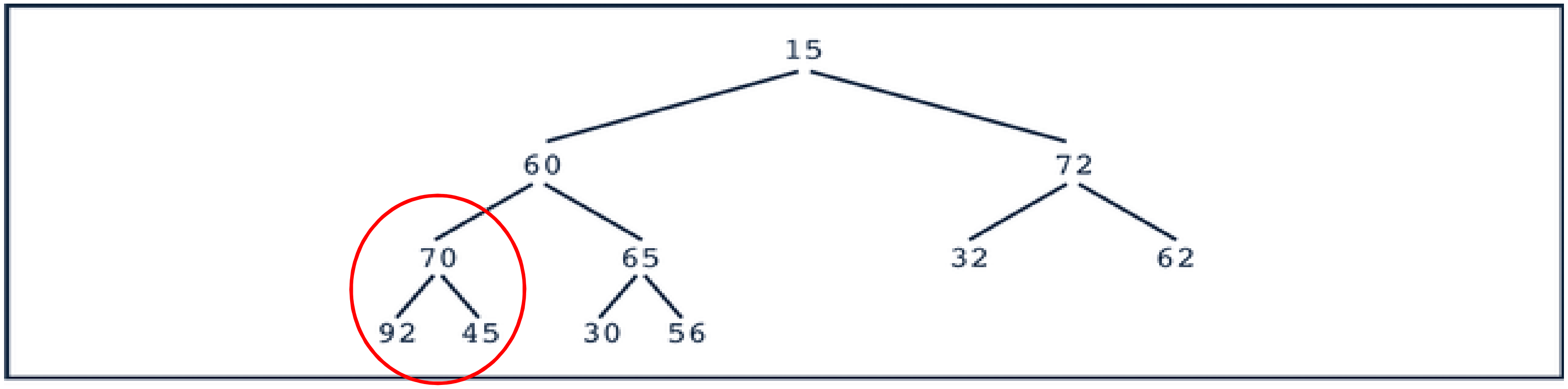
- **Build Heap**

The above list is not a heap. Let’s see how to build it into a heap

Building a Heap

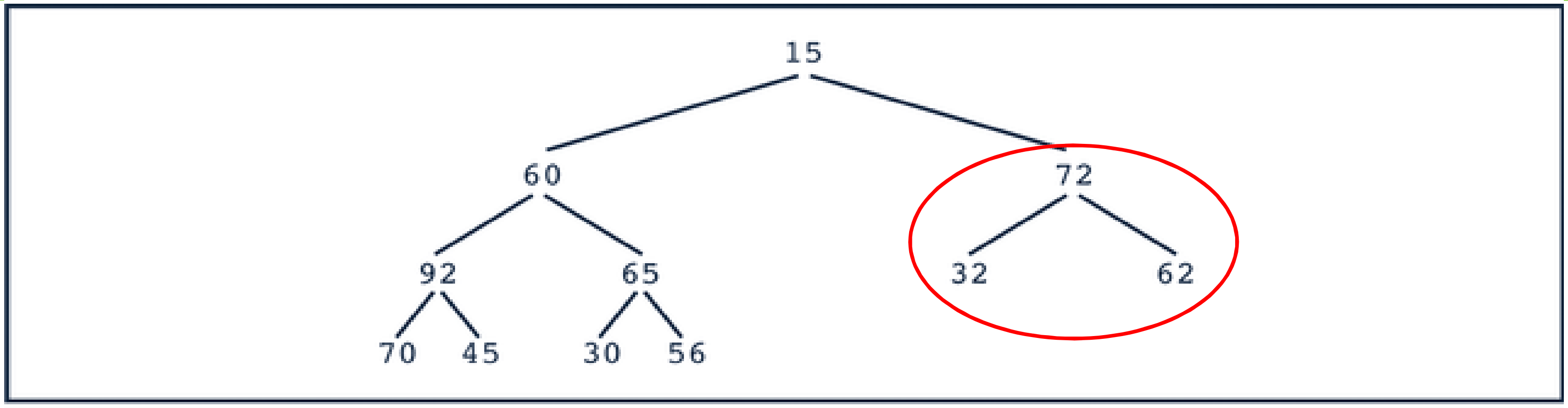


Complete binary tree corresponding to the list

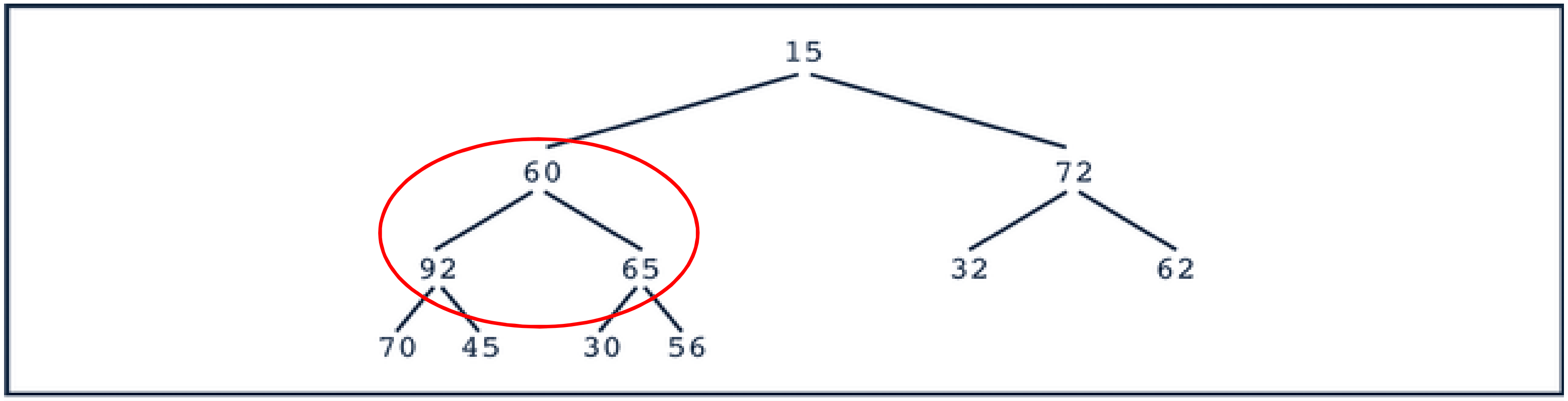


Binary tree after swapping 56 and 65

Building a Heap

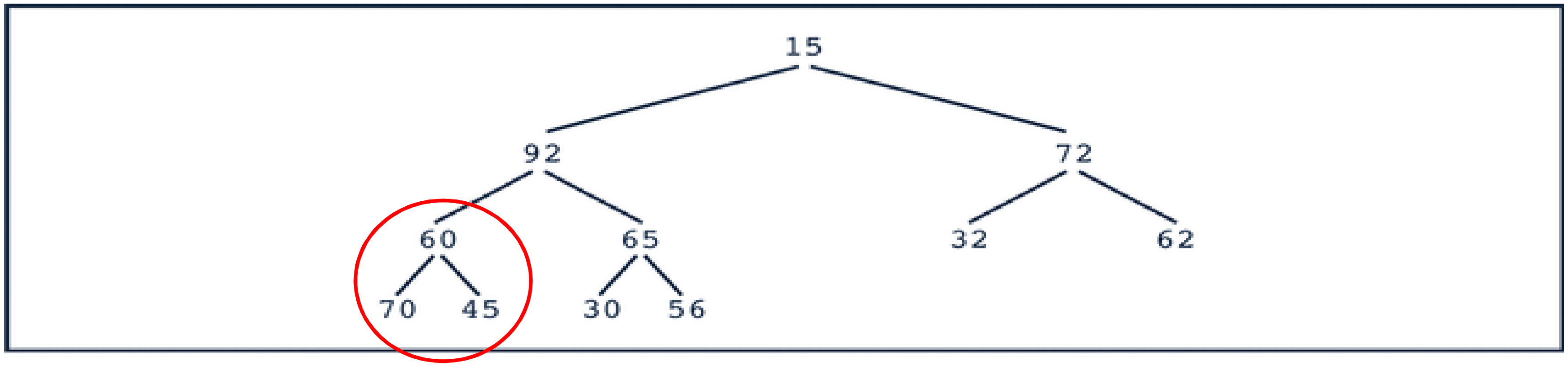


Binary tree after swapping 92 and 70

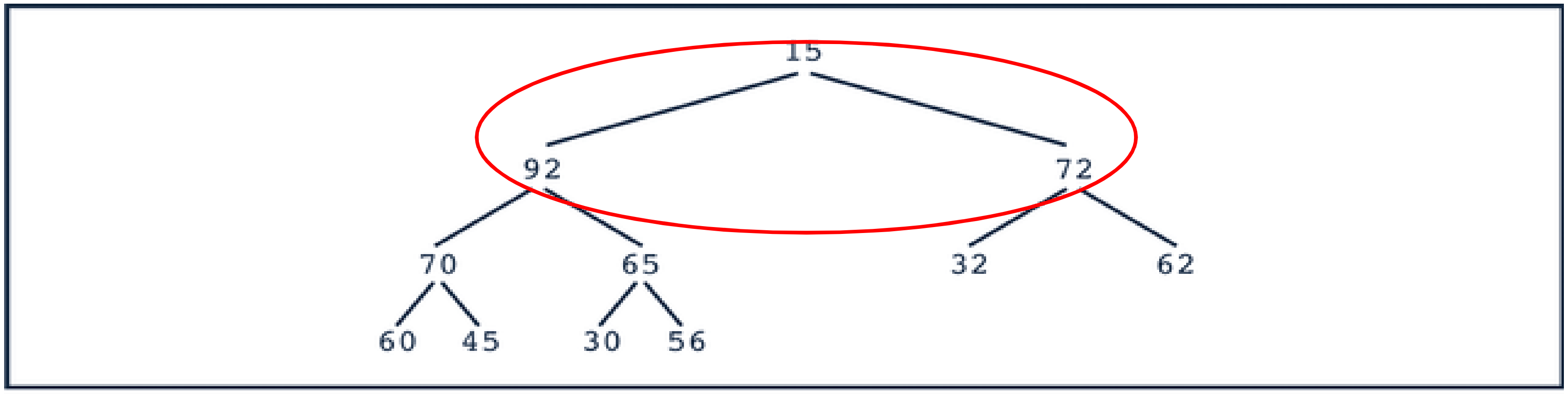


Binary tree after processing 72: no swapping required

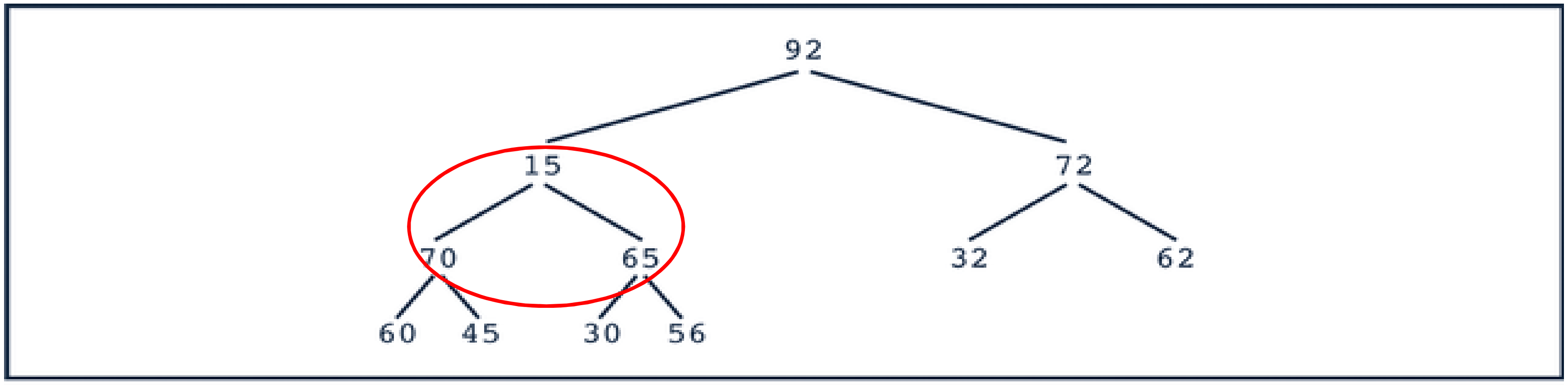
Building a Heap



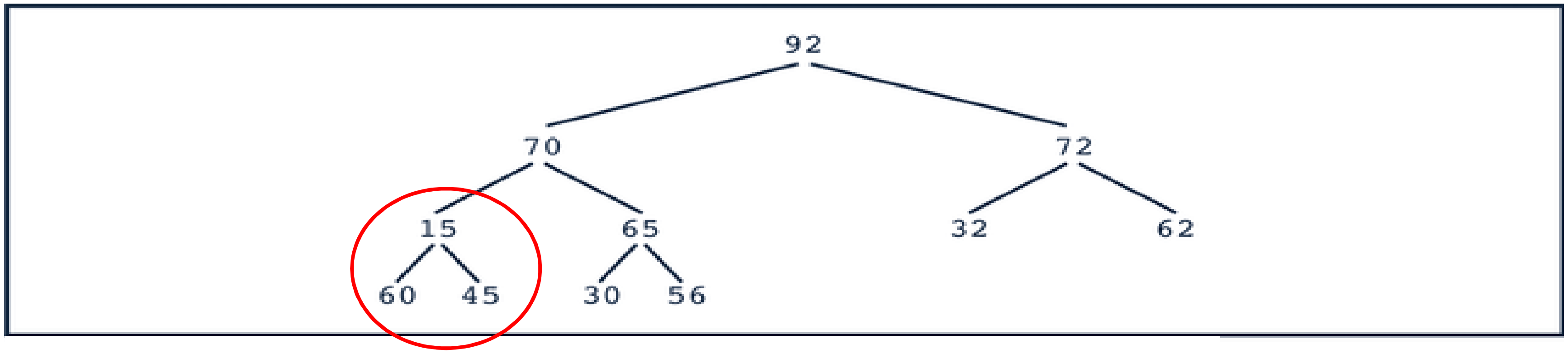
Binary tree after swapping 92 and 60



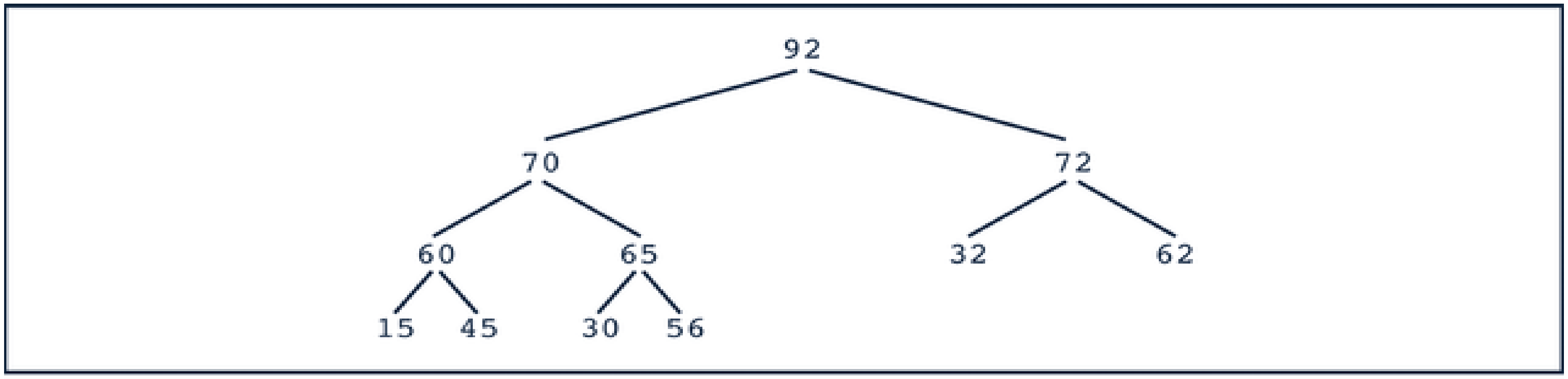
Binary tree after swapping 60 and 70



Binary tree after swapping 92 and 15



Binary tree after swapping 15 and 70



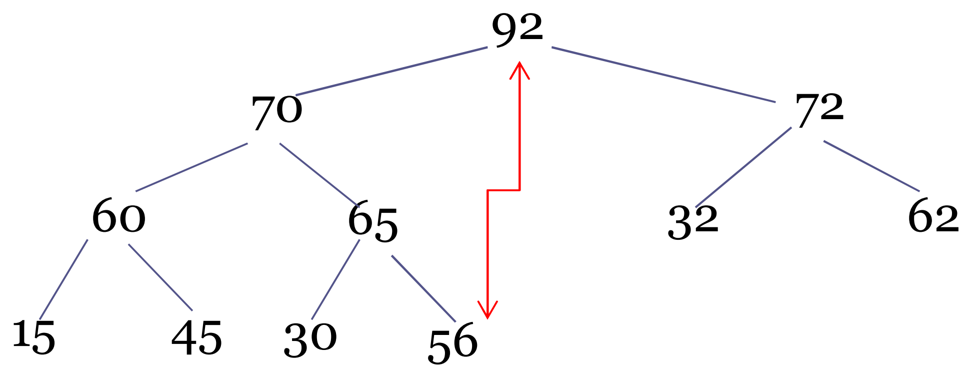
Binary tree after swapping 15 and 60

- **Now, the list becomes a heap**

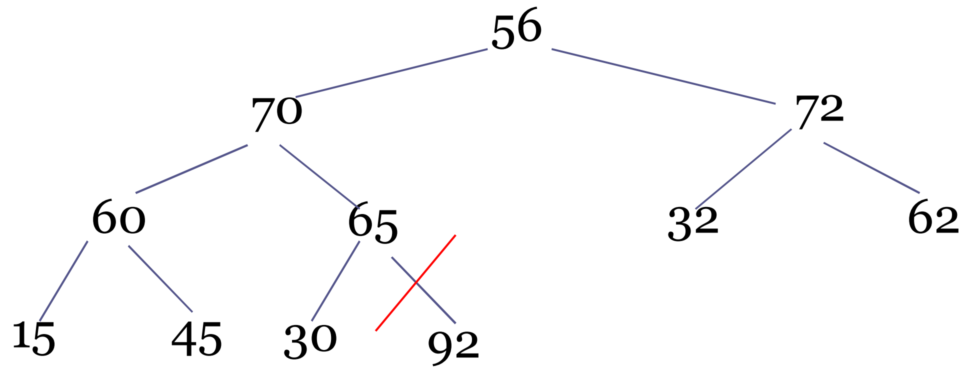
Heap: deleteMax

- Swap first element(Max) with last element
- Remove last element from heap
- Adjust the heap so that it satisfies heap property.

Heap: deleteMax

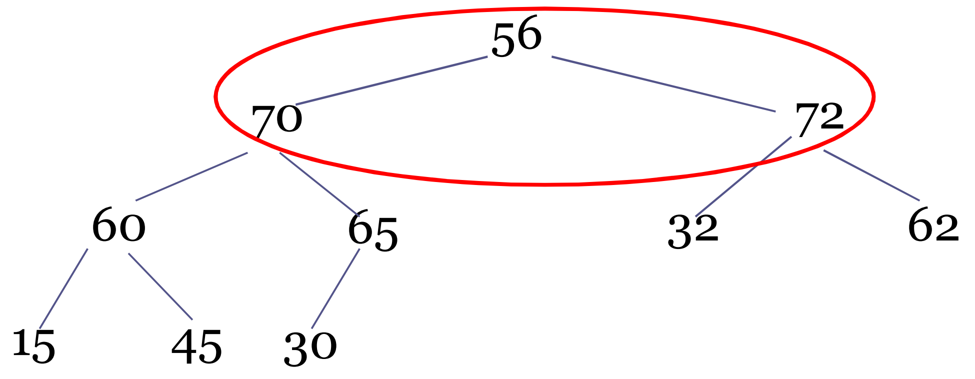


Swap first with last (92 with 56)

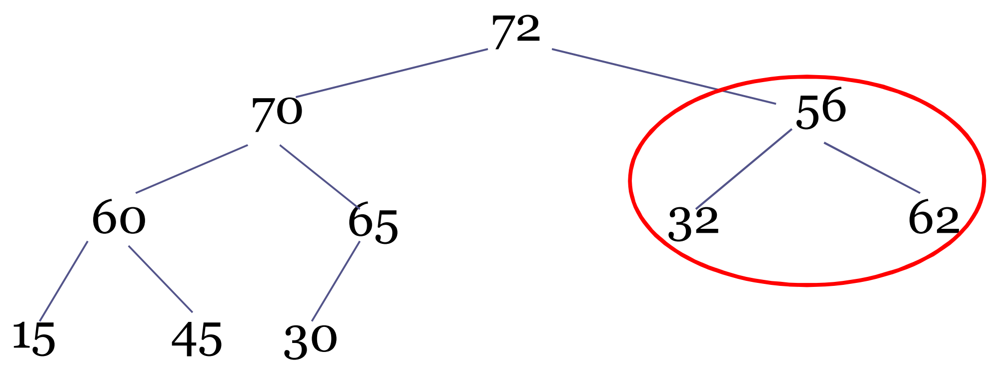


Remove last element(92)

Heap: deleteMax

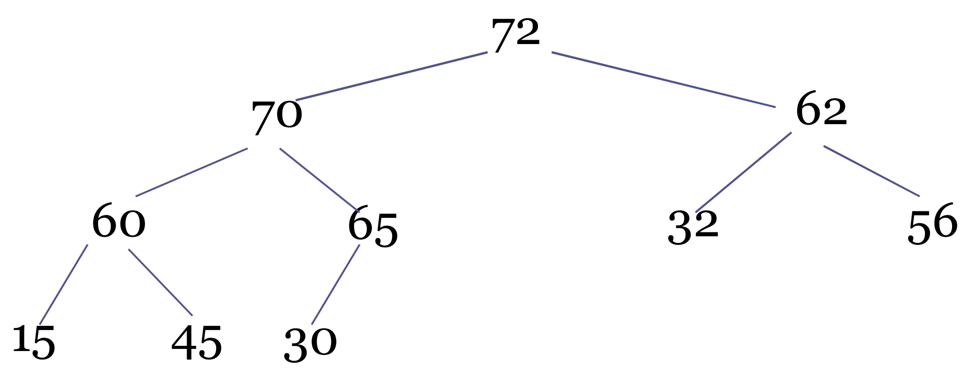


Heap property disturbed: Adjust it



Binary heap after swapping 56 with 72

Heap: deleteMax



Binary heap after swapping 56 with 62

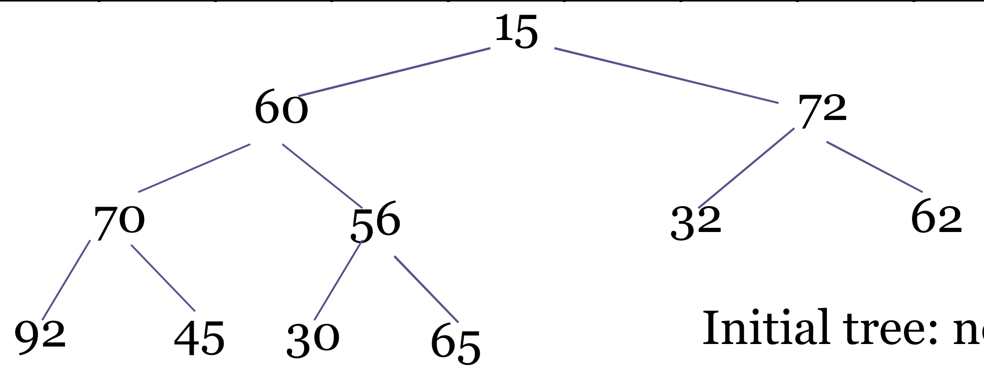
- Now it is again a Heap

Heap Sort

1. Convert initial array into a heap.
2. deleteMax : actually place largest at end of array.
3. repeat step 2 on reduced size heap.

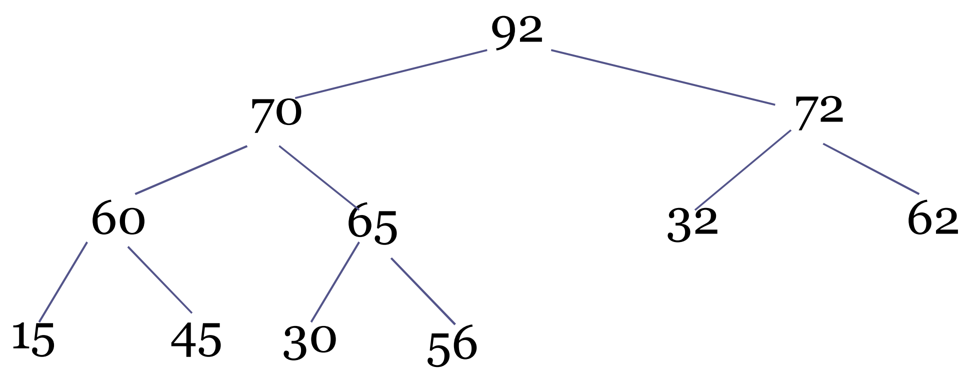
Heap Sort

0	1	2	3	4	5	6	7	8	9	10
15	**60**	**72**	**70**	**56**	**32**	**62**	**92**	**45**	**30**	**65**



Initial tree: not a heap

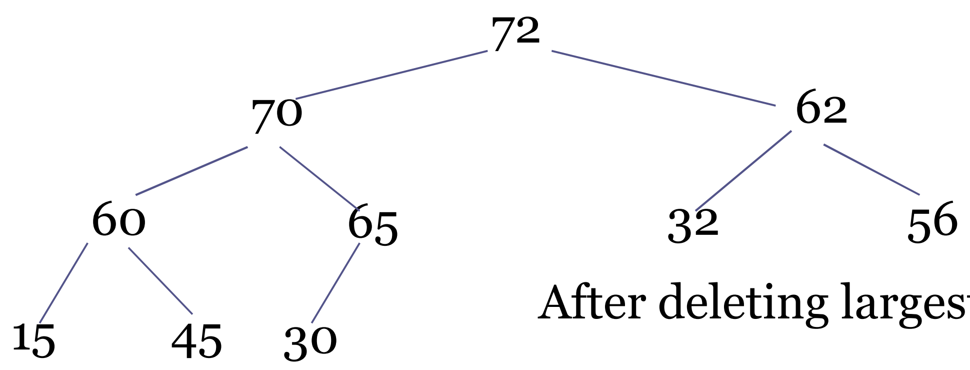
0	1	2	3	4	5	6	7	8	9	10
92	**70**	**72**	**60**	**65**	**32**	**62**	**15**	**45**	**30**	**56**



After converting list into heap

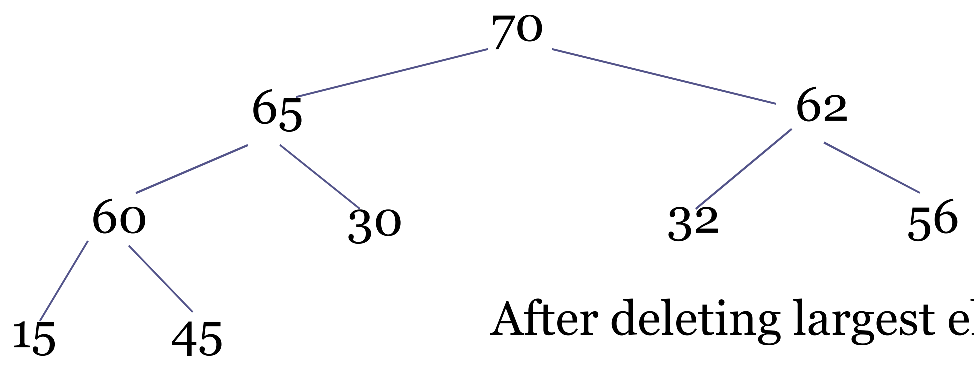
Heap Sort

0	1	2	3	4	5	6	7	8	9	10
72	70	62	60	65	32	56	15	45	30	92



After deleting largest element (92) from heap

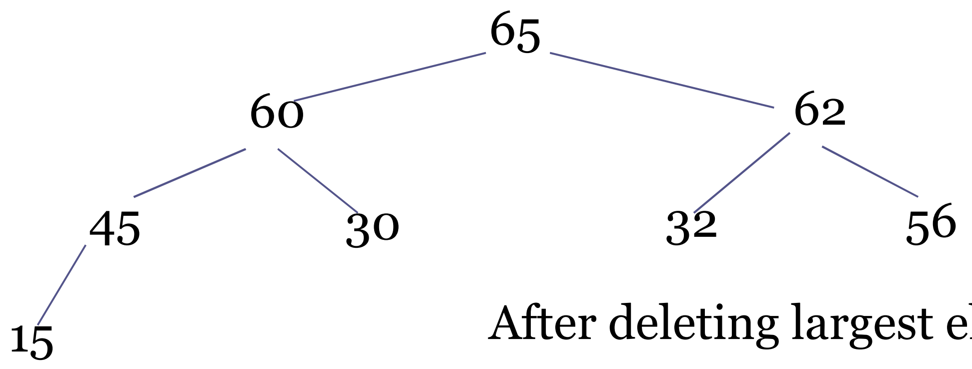
0	1	2	3	4	5	6	7	8	9	10
70	65	62	60	30	32	56	15	45	72	92



After deleting largest element (72) from heap

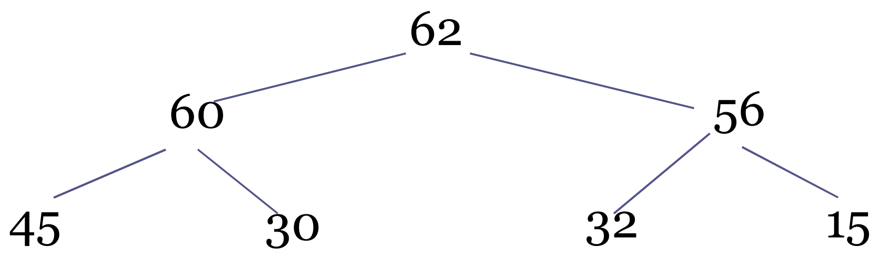
Heap Sort

0	1	2	3	4	5	6	7	8	9	10
65	60	62	45	30	32	56	15	70	72	92



After deleting largest element (70) from heap

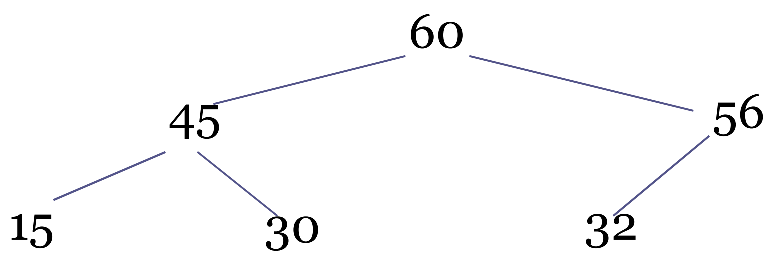
0	1	2	3	4	5	6	7	8	9	10
62	60	56	45	30	32	15	65	70	72	92



After deleting largest element (65) from heap

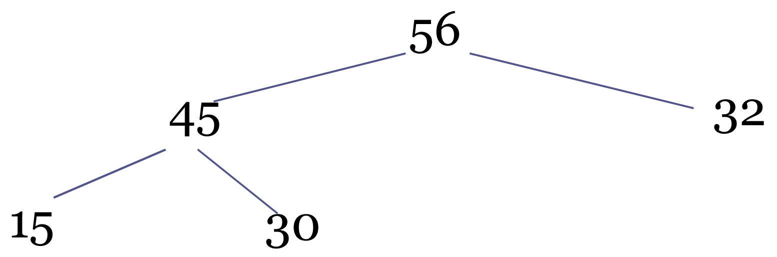
Heap Sort

0	1	2	3	4	5	6	7	8	9	10
60	**45**	**56**	**15**	**30**	**32**	**62**	**65**	**70**	**72**	**92**



After deleting largest element (62) from heap

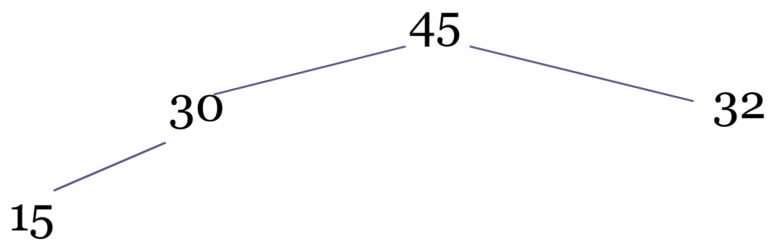
0	1	2	3	4	5	6	7	8	9	10
56	**45**	**32**	**15**	**30**	**60**	**62**	**65**	**70**	**72**	**92**



After deleting largest element (60) from heap

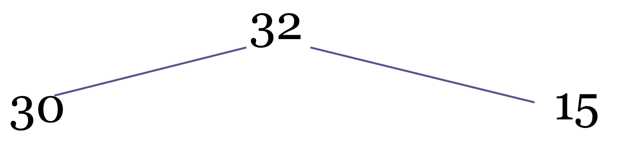
Heap Sort

0	1	2	3	4	5	6	7	8	9	10
45	**30**	**32**	**15**	**56**	**60**	**62**	**65**	**70**	**72**	**92**



After deleting largest element (56) from heap

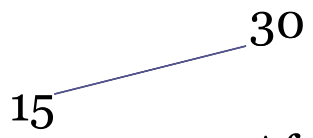
0	1	2	3	4	5	6	7	8	9	10
32	**30**	**15**	**45**	**56**	**60**	**62**	**65**	**70**	**72**	**92**



After deleting largest element (45) from heap

Heap Sort

0	1	2	3	4	5	6	7	8	9	10
30	15	32	45	56	60	62	65	70	72	92



After deleting largest element (32) from heap

0	1	2	3	4	5	6	7	8	9	10
15	30	32	45	56	60	62	65	70	72	92

15

After deleting largest element (30) from heap

0	1	2	3	4	5	6	7	8	9	10
15	30	32	45	56	60	62	65	70	72	92

After deleting largest element (15) from heap

Now array is sorted

Build initial Heap: Algorithm

```
Build_Heap(A[], N)
{
    FOR i=n/2 -1 DOWNTO 0 DO
    {
            Heapify(A,N, i);              // convert array into heap
    }
}

Heapify(A[], N,i);                        //Heap property is disturbed at node i, adjust it
{
            left=2*i+1;   right=2*i+2;
            IF (left < N   AND   A[left] > A[i]) THEN
                        largest = left; // find largest of two children
            ELSE
                        largest = i;
            IF (right <N  AND   A[right] > A[largest]) THEN
                        largest=right;
            IF largest NOT  = i) THEN
            {                                         //if needed  exchange parent with larger child
                        exchange  A[i] <-> A[largest]
                        Heapify(A,N,largest);        //now heap property of child is disturbed, adjust it
            }
}
```

Heap Sort : Algorithm

```
Heap_Sort(A[],N)
{
  FOR i=N-1 DOWNTO 1 DO
  {
        swap A[0] ←→ A[i]; //move largest at end
        N=N-1;        //remove last element from heap
        Heapify(A,N,0); //now adjust heap property of
                               first node
  }
}
```

Order O(N logN) algorithm