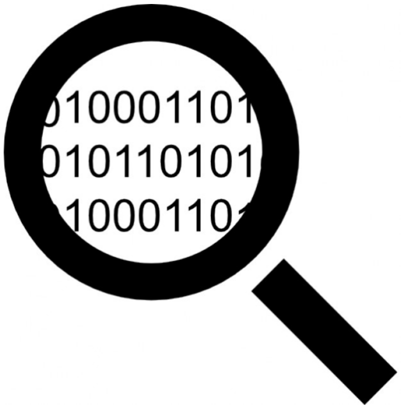
UNIVERSITY OF DIYALA

Data Structure

Lecture 8: Heaps, Searching

Instructor

Ali A. Al-Ani



Department of Computer Science
College of Science

UNIVERSITY OF DIYALA

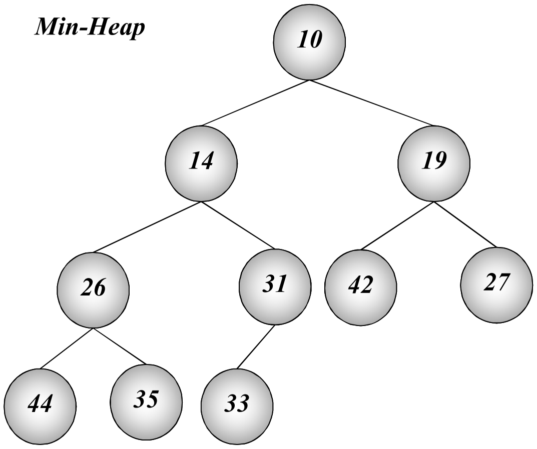
Heaps

- *A **Heap** is a special Tree-based data structure in which the tree is a **complete binary tree**. A heap is a binary tree **T** that stores a collection of elements with following properties.*
  1. ***It's a complete tree** (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.*
  2. ***A Binary Heap is either Min Heap or Max Heap**.*
     - A. ***Max-Heap:** In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.*
     - B. ***Min-Heap:** In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree..*

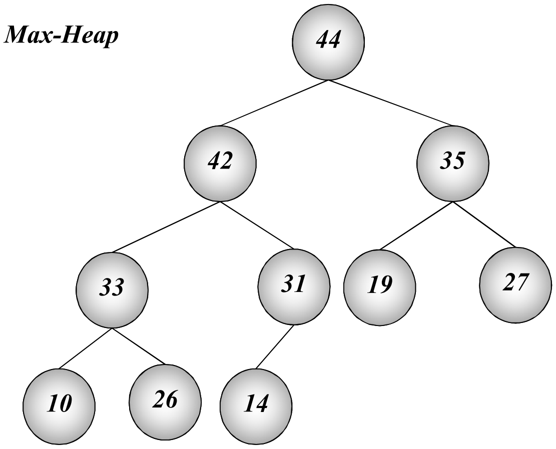
Department of Computer Science
College of Science

Heaps

- *Min-Heap*



- *Max-Heap*



Heaps: Construction Algorithm

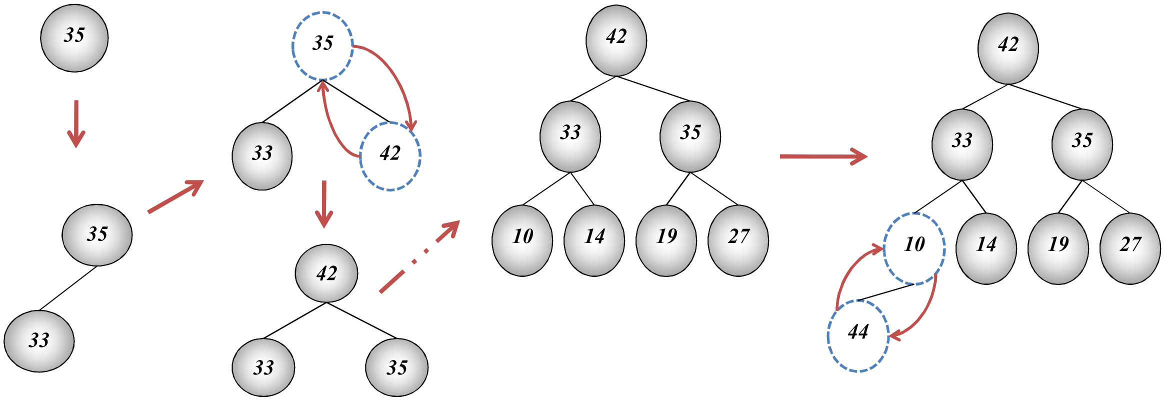
- ***Max Heap Construction Algorithm:*** *in Max Heap construction algorithm, we expect the value of the* ***parent node*** *to be* ***greater than*** *that of the* ***child node****. We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property.*

    1. ***Create a new node at the end of heap.***
    2. ***Assign new value to the node.***
    3. ***Compare the value of this child node with its parent.***
    4. ***If value of parent is less than child, then swap them.***
    5. ***Repeat step 3 & 4 until Heap property holds.***

UNIVERSITY OF DIYALA

Heaps: Construction Algorithm

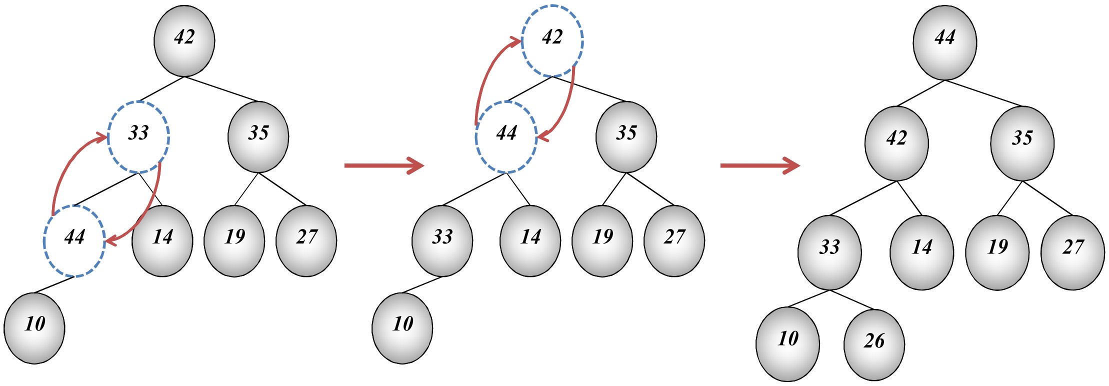
- ***Ex:*** *Using the following data items to construct the max-heap tree.* ***35, 33, 42, 10, 14, 19, 27, 44, 26***



UNIVERSITY OF DIYALA

Heaps: Construction Algorithm

- ***Ex:*** *Using the following data items to construct the max-heap tree.* ***35, 33, 42, 10, 14, 19, 27, 44, 26***



UNIVERSITY OF DIYALA

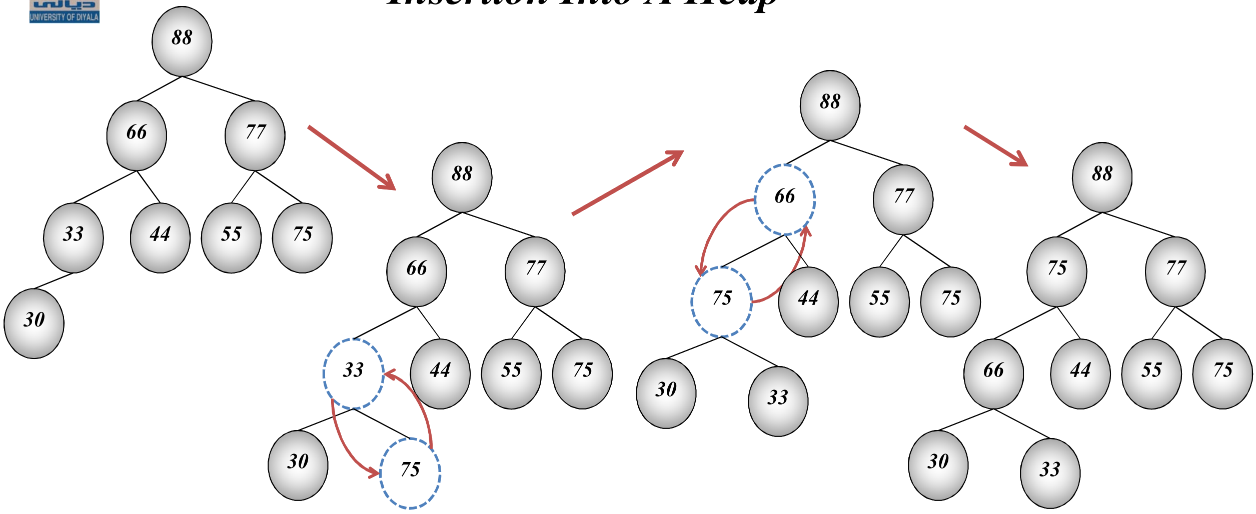
Insertion Into A Heap

- *Elements are inserted into a heap next to its right-most leaf at the bottom level. Then the heap property is restored by percolating the new element up the tree until it is no longer "older" (i.e., its key is greater) than its parent. On each iteration, the child is swapped with its parent.*
- *The following example shows how the key **75** would be inserted into the heap. The element **75** is added to the tree as a new last leaf. Then it is swapped with its parent element **33** because **75 > 33**. Then it is swapped with its parent element **66** because **75 > 66**. Now the heap property has been restored because the new element **75** is less than its parent and greater than its children.*

Department of Computer Science
College of Science

UNIVERSITY OF DIYALA

Insertion Into A Heap



Department of Computer Science
College of Science

UNIVERSITY OF DIYALA

Heaps: Deletion Algorithm

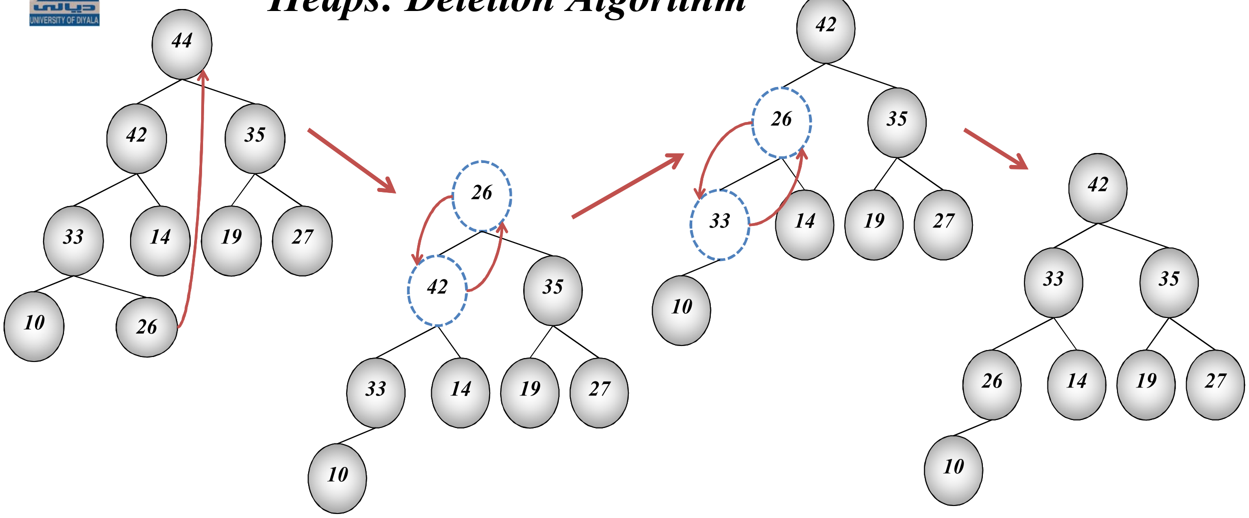
- ***Max Heap Deletion Algorithm:*** *Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value, and the steps are follow:*

  1. ***Remove root node.***
  2. ***Move the last element of last level to root.***
  3. ***Compare the value of this child node with its parent.***
  4. ***If value of parent is less than child, then swap them.***
  5. ***Repeat step 3 & 4 until Heap property holds.***

Department of Computer Science
College of Science

UNIVERSITY OF DIYALA

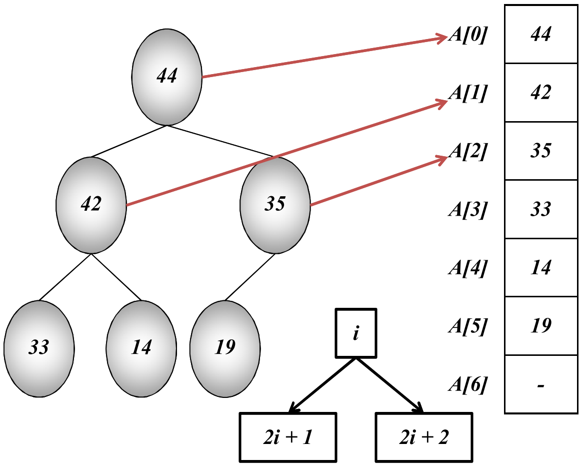
Heaps: Deletion Algorithm



Department of Computer Science
College of Science

Representing Heaps as Arrays

- ***Linear representation of a Heaps utilizes one-dimensional*** *array of size* 2^{h+1} ***- 1****. Consider the following Heap. To represent this Heap, we need an array of size* 2^{2+1} ***- 1 =*** *7, The Heap is represented as follows* ***A[7]****.*



A[0]	44
A[1]	42
A[2]	35
A[3]	33
A[4]	14
A[5]	19
A[6]	-

Heaps: Applications

- ***The heap data structure has many applications.***

1. *To quickly find the smallest and largest element from a collection of items or array.*
2. ***Heap sort:*** *One of the best sorting methods being in-place and with no quadratic worst-case scenarios.*
3. *In the implementation of* ***Priority queue*** *in graph algorithms like Dijkstra's algorithm and Prim's algorithm.* ***priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority.***

UNIVERSITY OF DIYALA

Heap Sort

- *Heap Sort is a popular and efficient sorting algorithm in computer programming. Heap sort is a comparison based sorting technique based on Heap data structure, And the steps of sorting as follow:*

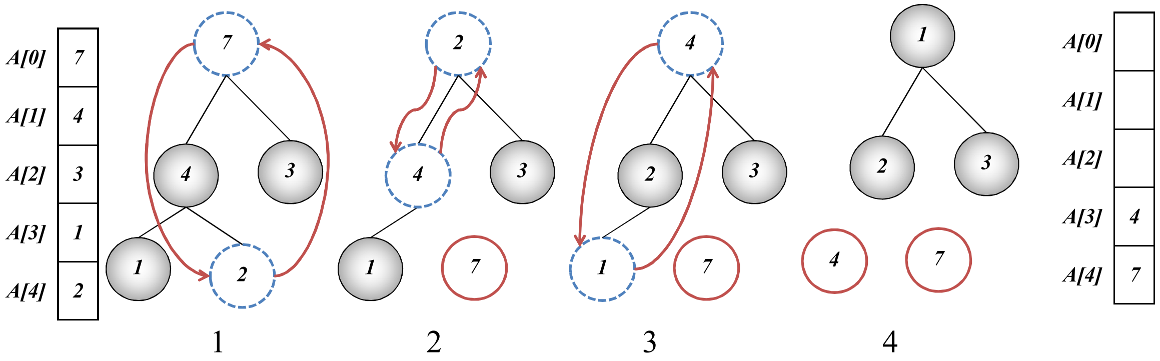
  1. ***Build a max-heap from the array***
  2. ***Swap the root (the maximum element) with the last element in the array***
  3. ***“Discard” this last node by decreasing the heap size***
  4. ***Call Max- Heapify on the new root***
  5. ***Repeat this process until only one node remains***

Department of Computer Science
College of Science

UNIVERSITY OF DIYALA

Heap Sort

Ex: *Given an array of 6 elements:* ***7, 4, 3, 1, 2*** *sort it in ascending order using heap sort. Consider the values of the elements as priorities and build the heap tree.*

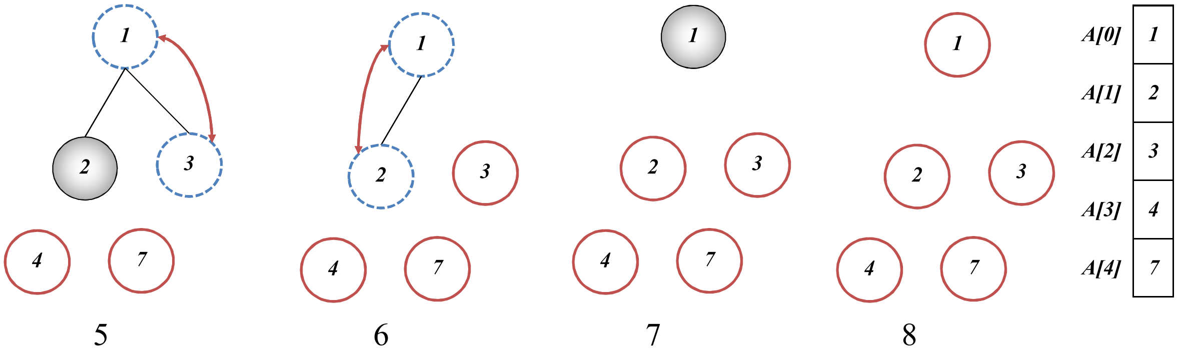


Department of Computer Science
College of Science

UNIVERSITY OF DIYALA

Heap Sort

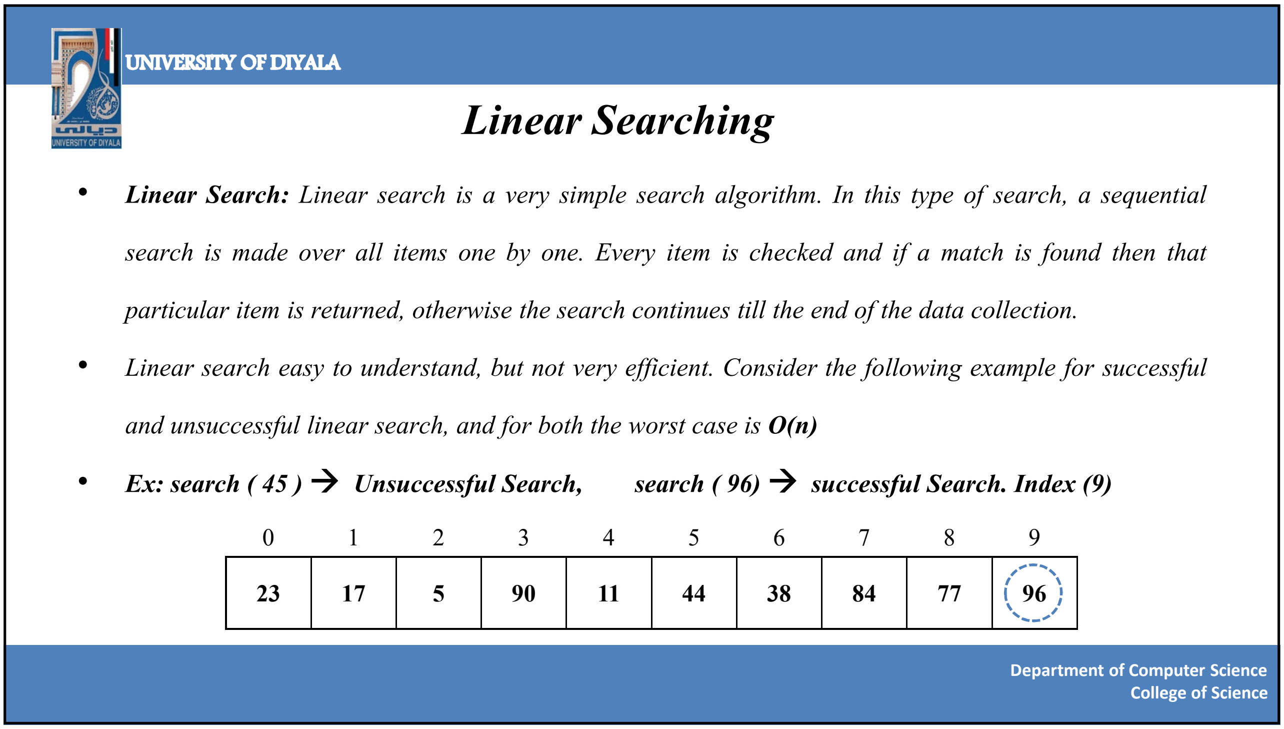
Ex: *Given an array of 6 elements:* ***7, 4, 3, 1, 2*** *sort it in ascending order using heap sort. Consider the values of the elements as priorities and build the heap tree.*



UNIVERSITY OF DIYALA

Searching

- ***Searching*** *is the process of finding a target element among a group of items (the search pool), or determining that it isn't there. Here's the problem statement:*
  - *Given a value X, return the index of X in the array, if such X exists. Otherwise, return NOT_FOUND (-1). Assume there are no duplicate entries in the array.*
- *The number of comparisons the algorithms make to analyze their performance. The ideal searching algorithm will make the least possible number of comparisons to locate the desired data. Two separate performance analyses are normally done, one for successful search and another for unsuccessful search.*



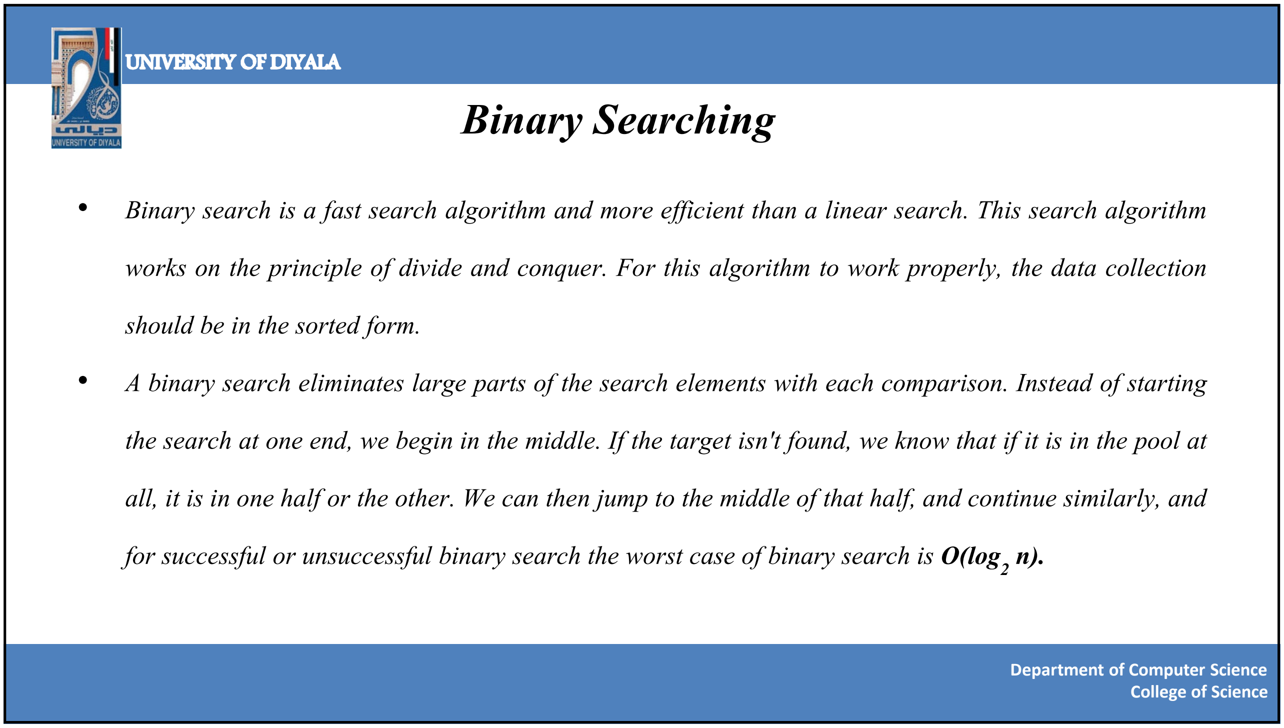
UNIVERSITY OF DIYALA

Linear Searching

- ***Linear Search:*** *Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.*
- *Linear search easy to understand, but not very efficient. Consider the following example for successful and unsuccessful linear search, and for both the worst case is* ***O(n)***
- ***Ex: search (45) → Unsuccessful Search, search (96) → successful Search. Index (9)***

0	1	2	3	4	5	6	7	8	9
23	**17**	**5**	**90**	**11**	**44**	**38**	**84**	**77**	**96**

Department of Computer Science
College of Science

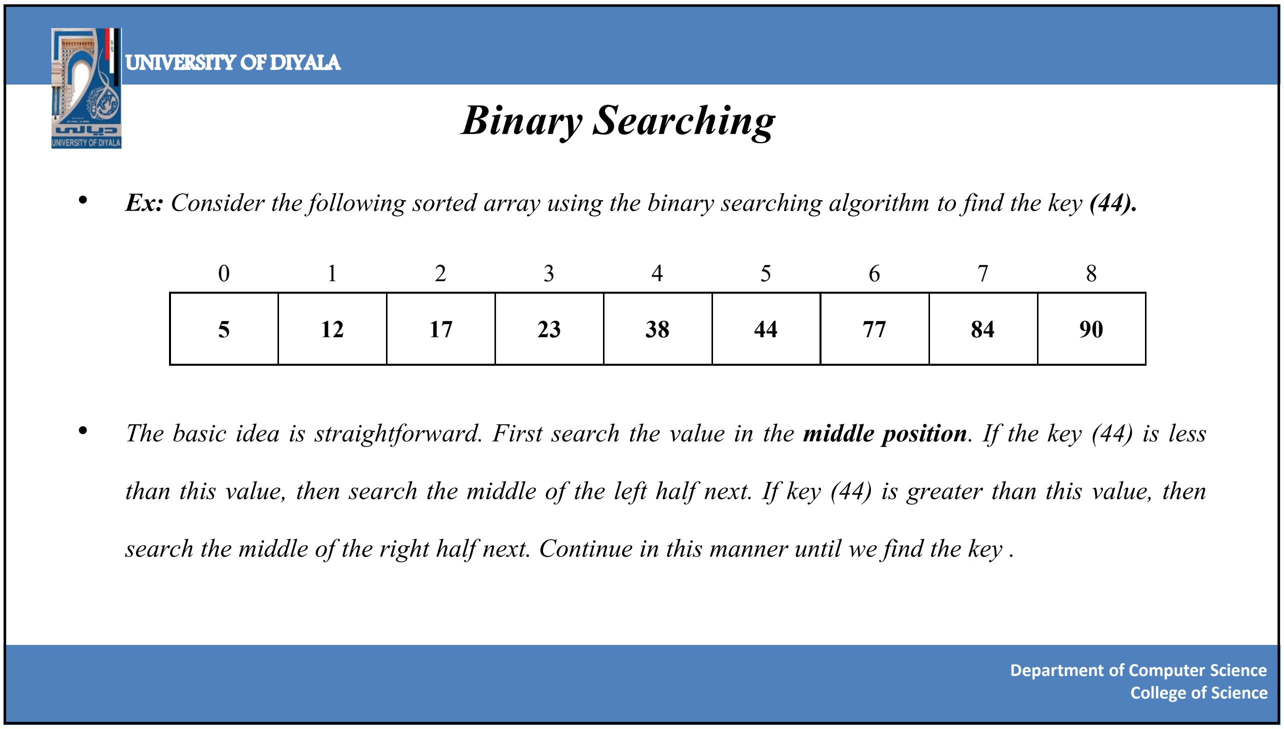


UNIVERSITY OF DIYALA

Binary Searching

- *Binary search is a fast search algorithm and more efficient than a linear search. This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.*
- *A binary search eliminates large parts of the search elements with each comparison. Instead of starting the search at one end, we begin in the middle. If the target isn't found, we know that if it is in the pool at all, it is in one half or the other. We can then jump to the middle of that half, and continue similarly, and for successful or unsuccessful binary search the worst case of binary search is* ***$O(\log_2 n)$.***

Department of Computer Science
College of Science



UNIVERSITY OF DIYALA

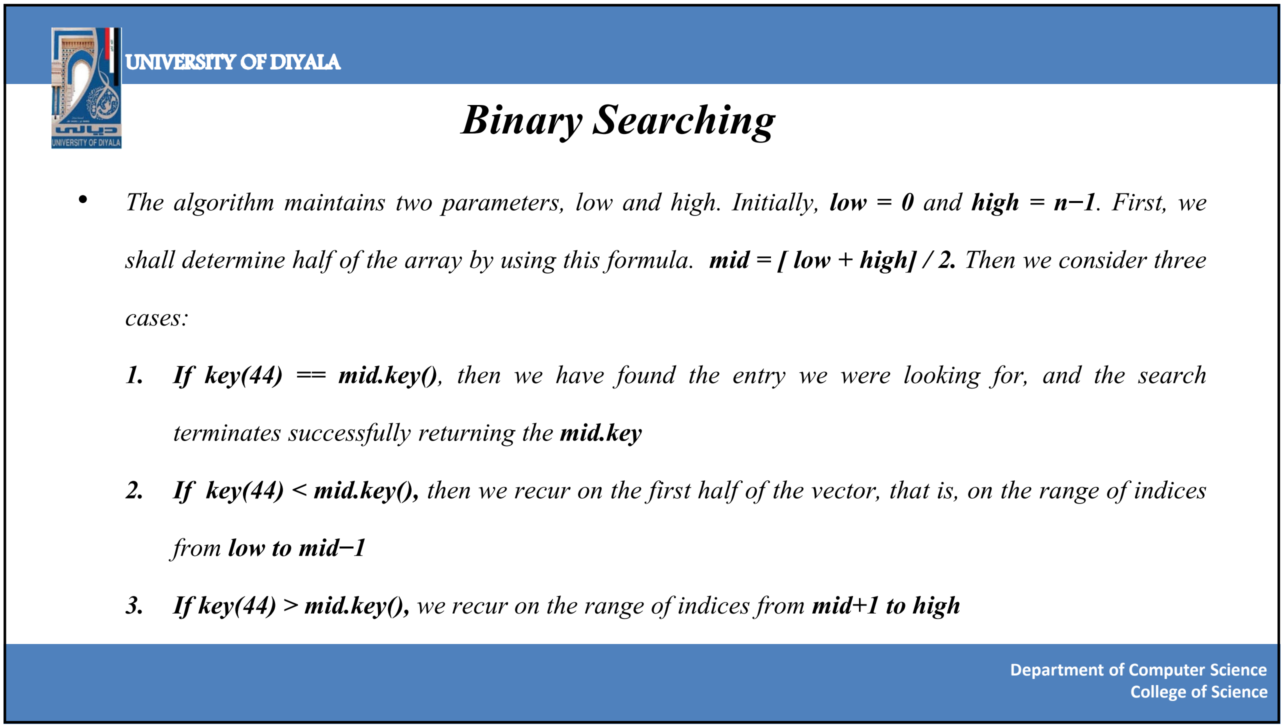
Binary Searching

- ***Ex:*** *Consider the following sorted array using the binary searching algorithm to find the key* ***(44).***

0	1	2	3	4	5	6	7	8
5	**12**	**17**	**23**	**38**	**44**	**77**	**84**	**90**

- *The basic idea is straightforward. First search the value in the* ***middle position****. If the key (44) is less than this value, then search the middle of the left half next. If key (44) is greater than this value, then search the middle of the right half next. Continue in this manner until we find the key .*

Department of Computer Science
College of Science



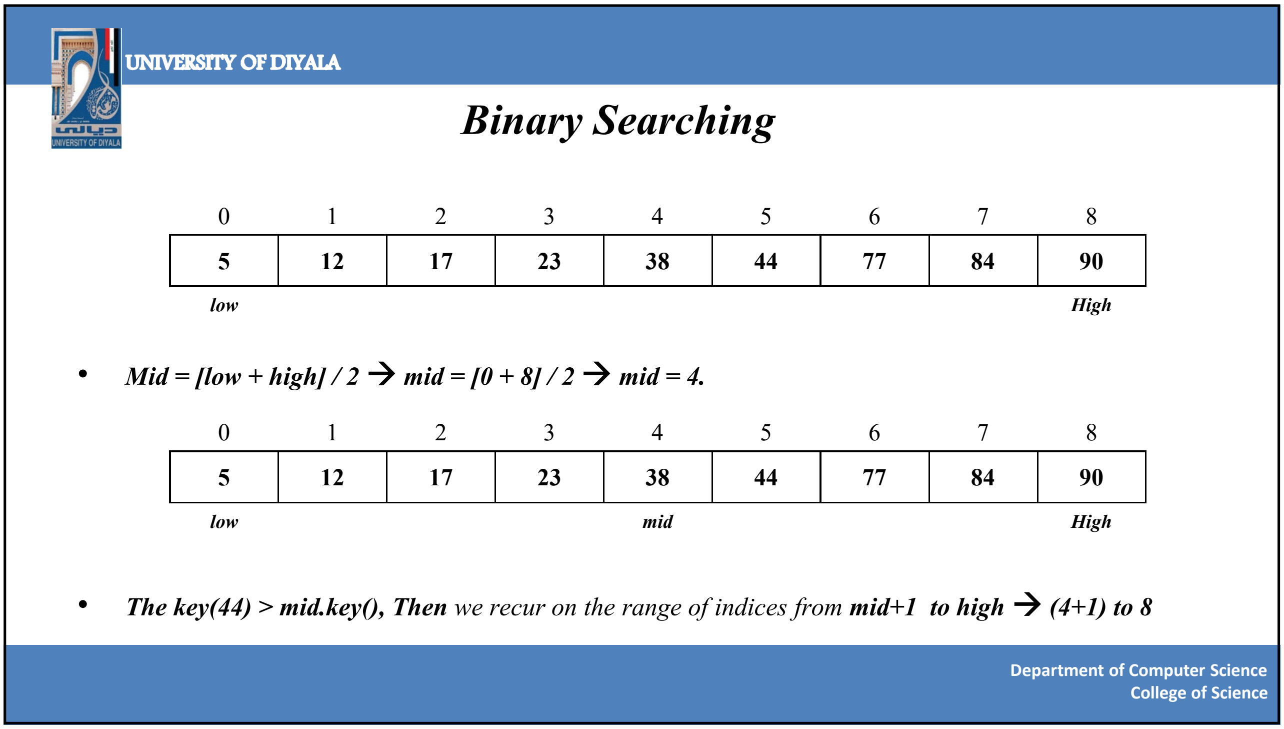
UNIVERSITY OF DIYALA

Binary Searching

- *The algorithm maintains two parameters, low and high. Initially,* ***low = 0*** *and* ***high = n−1****. First, we shall determine half of the array by using this formula.* ***mid = [low + high] / 2.*** *Then we consider three cases:*

1. ***If key(44) == mid.key()****, then we have found the entry we were looking for, and the search terminates successfully returning the* ***mid.key***
2. ***If key(44) < mid.key(),*** *then we recur on the first half of the vector, that is, on the range of indices from* ***low to mid−1***
3. ***If key(44) > mid.key(),*** *we recur on the range of indices from* ***mid+1 to high***

Department of Computer Science
College of Science



UNIVERSITY OF DIYALA

Binary Searching

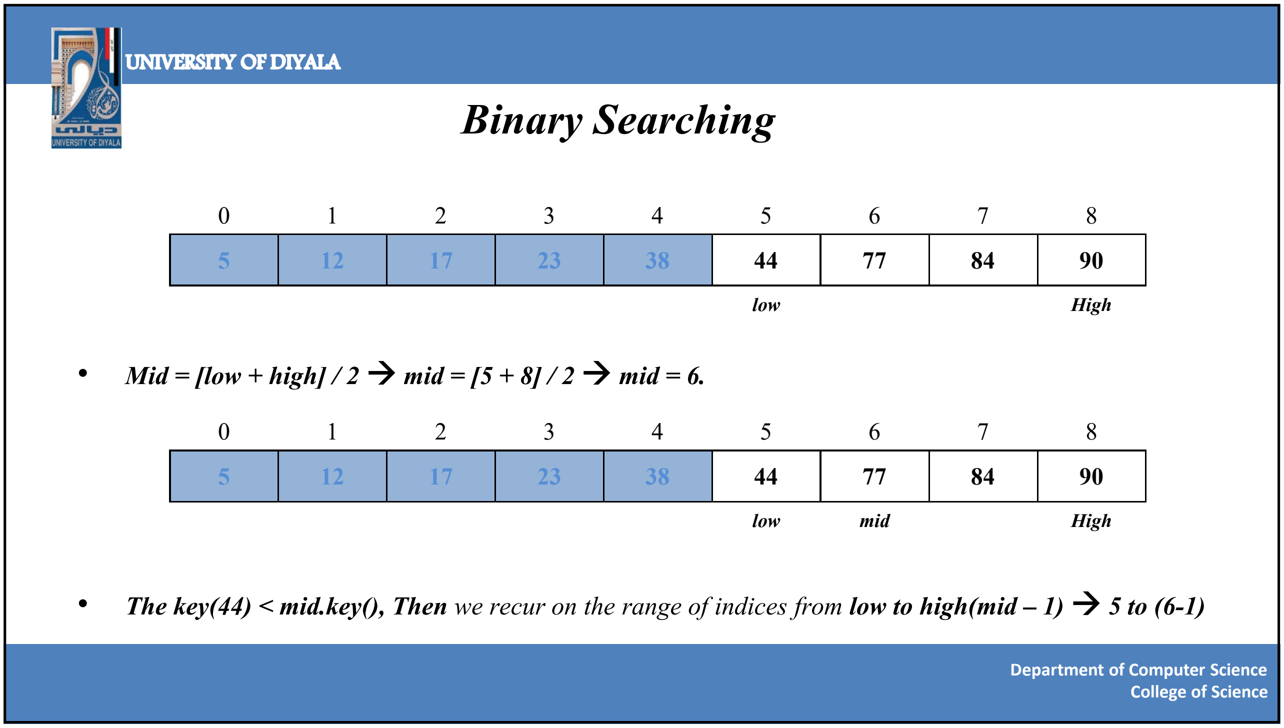
0	1	2	3	4	5	6	7	8
5	**12**	**17**	**23**	**38**	**44**	**77**	**84**	**90**
low								*High*

- ***Mid = [low + high] / 2 → mid = [0 + 8] / 2 → mid = 4.***

0	1	2	3	4	5	6	7	8
5	**12**	**17**	**23**	**38**	**44**	**77**	**84**	**90**
low				*mid*				*High*

- ***The key(44) > mid.key(), Then** we recur on the range of indices from **mid+1 to high → (4+1) to 8***

Department of Computer Science
College of Science



UNIVERSITY OF DIYALA

Binary Searching

0	1	2	3	4	5	6	7	8
5	12	17	23	38	**44**	**77**	**84**	**90**
					low			*High*

- ***Mid = [low + high] / 2 → mid = [5 + 8] / 2 → mid = 6.***

0	1	2	3	4	5	6	7	8
5	12	17	23	38	**44**	**77**	**84**	**90**
					low	*mid*		*High*

- ***The key(44) < mid.key(), Then** we recur on the range of indices from **low to high(mid – 1) → 5 to (6-1)***

Department of Computer Science
College of Science

Binary Searching

0	1	2	3	4	5	6	7	8
5	12	17	23	38	**44**	77	84	90
					Low, high			

- *Mid = [low + high] / 2 → mid = [5 + 5] / 2 → mid = 5.*

0	1	2	3	4	5	6	7	8
5	12	17	23	38	**44**	77	84	90
					Low, high, *Mid*			

- *The key(44) == mid.key(), we find the key at position 5 Successful Search*

The End