Sorting

COMP 251

Overview

- Recall that there are two basic searching algorithms; namely, linear (sequential) and binary.
- The difference between is fundamentally related to whether or not the input list is sorted or not.
- Thus, we may consider two fundamental tasks of **searching** and **sorting**.
- We have looked at searching. Now, let us focus on sorting.

Overview

Consider the following bit of review:

- ***Arrays*** and ***linked lists*** are two basic data structures used to store information.
- We may wish to **search**, **insert** or **delete** records in a database based on a key value.
- Let us examine the performance of these operations on arrays and linked lists.

https://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_man.pdf

Arrays

Figure 1-1 shows an array, seven elements long, containing numeric values.

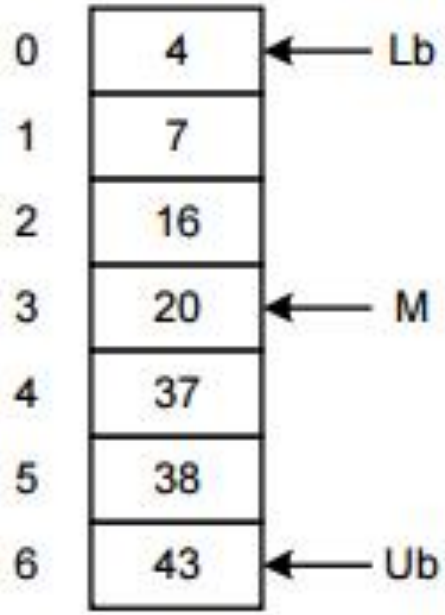


Figure 1-1: An Array

https://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_man.pdf

Arrays

Figure 1-1 shows an array, seven elements long, containing numeric values. To search the array *sequentially*, we may use the algorithm in Figure 1-2. The maximum number of comparisons is 7, and occurs when the key we are searching for is in *A*[6].

```
int function SequentialSearch (Array A , int Lb , int Ub , int Key );
    begin
    for i = Lb to Ub do
        if A [ i ] = Key then
            return i ;
        return –1;
    end;
```

Figure 1-2: Sequential Search

https://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_man.pdf

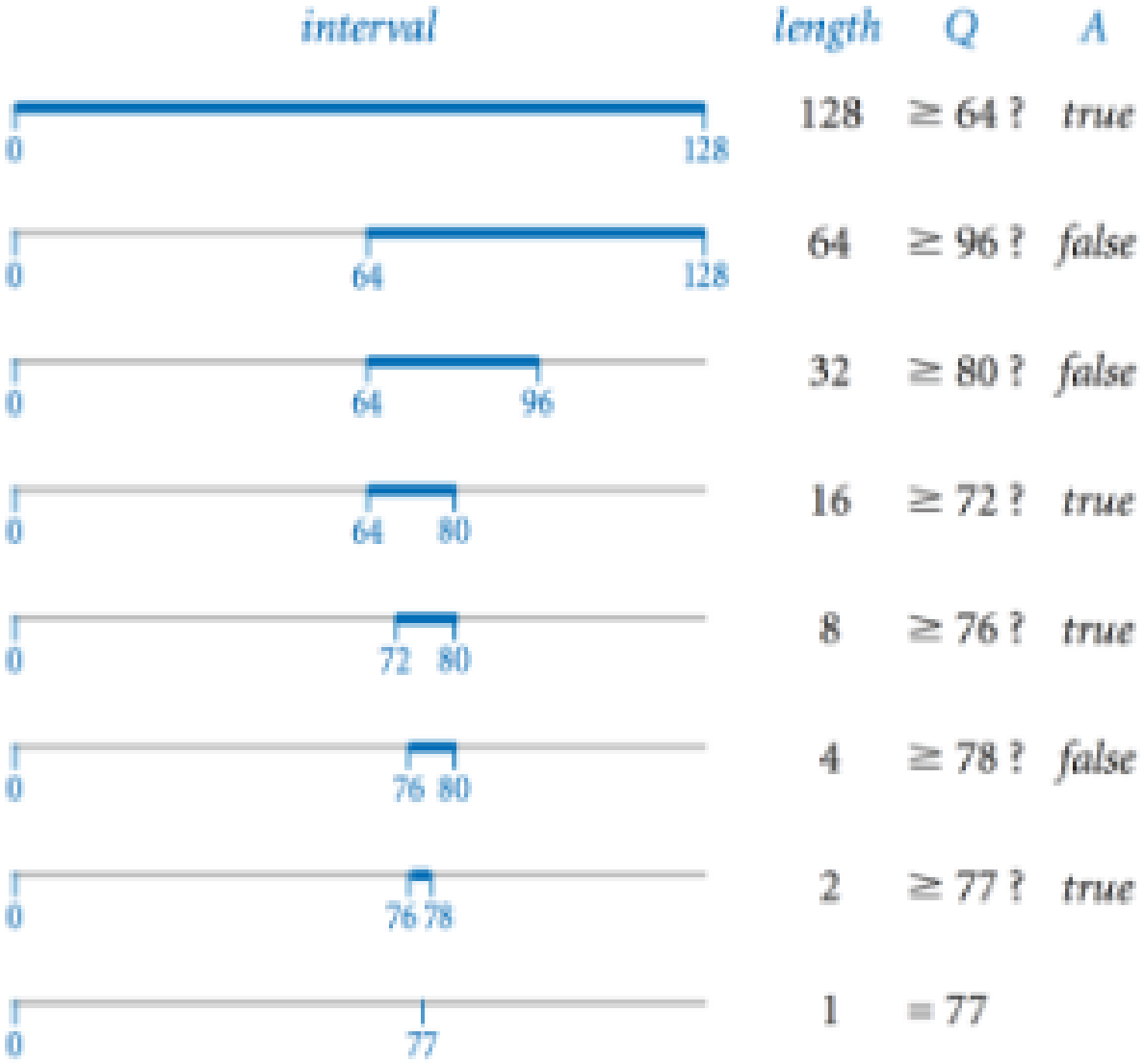
```
int function BinarySearch (Array A , int Lb , int Ub , int Key );
    begin
    do forever
        M = ( Lb + Ub )/2;
        if ( Key < A[M]) then
            Ub = M – 1;
        else if (Key > A[M]) then
            Lb = M + 1;
        else
            return M ;
        if (Lb > Ub) then
            return –1;
    end;
```

Figure 1-3: Binary Search

If the data is sorted, a *binary* search may be done (Figure 1-3). Variables *Lb* and *Ub* keep track of the *lower bound* and *upper bound* of the array, respectively. We begin by examining the middle element of the array. If the key we are searching for is less than the middle element, then it must reside in the top half of the array. Thus, we set *Ub* to ($M - 1$). This restricts our next iteration through the loop to the top half of the array. In this way, each iteration halves the size of the array to be searched. For example, the first iteration will leave 3 items to test. After the second iteration, there will be one item left to test. Therefore it takes only three iterations to find any number.

https://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_man.pdf

Binary Search



http://introcs.cs.princeton.edu/java/42sort/

This is a powerful method. Given an array of 1023 elements, we can narrow the search to 511 elements in one comparison. After another comparison, and we're looking at only 255 elements. In fact, we can search the entire array in only 10 comparisons.

https://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_man.pd

This is a powerful method. Given an array of 1023 elements, we can narrow the search to 511 elements in one comparison. After another comparison, and we're looking at only 255 elements. In fact, we can search the entire array in only 10 comparisons.

In addition to searching, we may wish to insert or delete entries. Unfortunately, an array is not a good arrangement for these operations. For example, to insert the number 18 in Figure 1-1, we would need to shift $A[3]\ldots A[6]$ down by one slot. Then we could copy number 18 into $A[3]$. A similar problem arises when deleting numbers. To improve the efficiency of insert and delete operations, linked lists may be used.

https://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_man.pd

Linked Lists

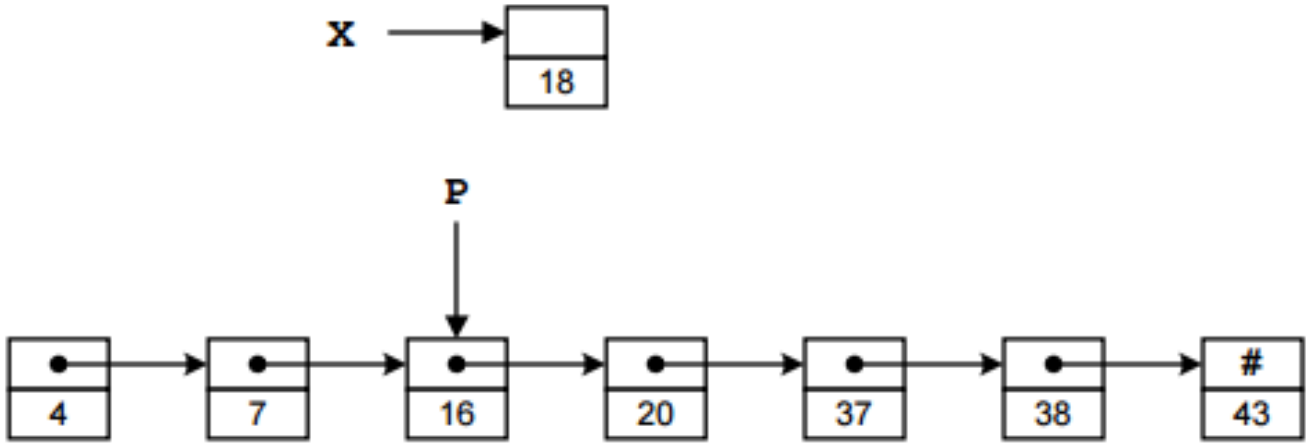


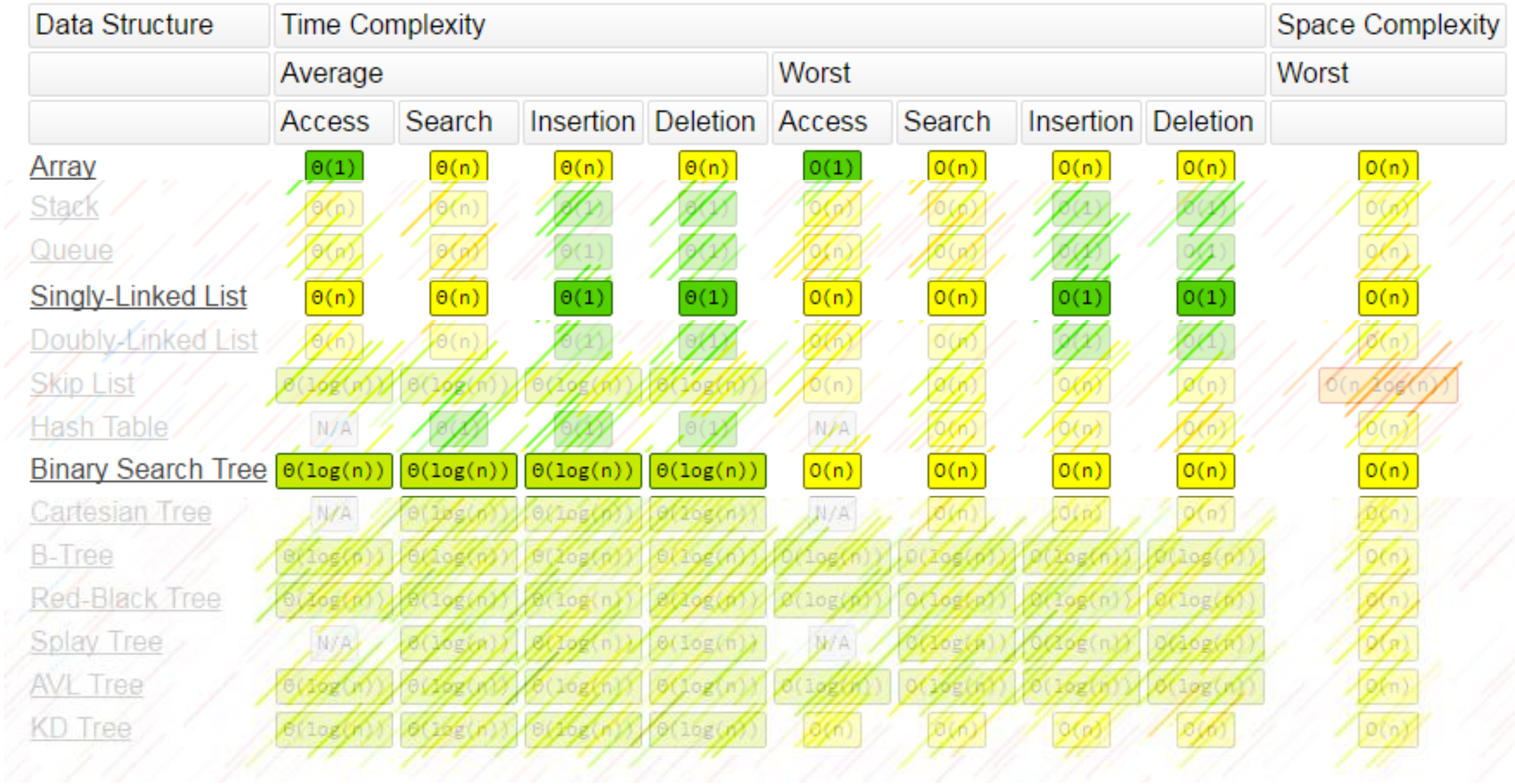
Figure 1-4: A Linked List

In Figure 1-4 we have the same values stored in a linked list. Assuming pointers **X** and **P**, as shown in the figure, value 18 may be inserted as follows:

```
X->Next = P->Next;
P->Next = X;
```

Insertion and deletion operations are very efficient using linked lists. You may be wondering how pointer **P** was set in the first place. Well, we had to do a sequential search to find the insertion point **X**. Although we improved our performance for insertion/deletion, it was done at the expense of search time.

https://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_man.pd



Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

n	$\lg n$	$n \lg n$	$n^{1.25}$	n^2
1	0	0	1	1
16	4	64	32	256
256	8	2,048	1,024	65,536
4,096	12	49,152	32,768	16,777,216
65,536	16	1,048,565	1,048,476	4,294,967,296
1,048,476	20	20,969,520	33,554,432	1,099,301,922,576
16,775,616	24	402,614,784	1,073,613,825	281,421,292,179,456

Table 1-1: Growth Rates

Table 1-1 illustrates growth rates for various functions. A growth rate of $O(\lg n)$ occurs for algorithms similar to the binary search. The lg (logarithm, base 2) function increases by one when n is doubled. Recall that we can search twice as many items with one more comparison in the binary search. Thus the binary search is a $O(\lg n)$ algorithm.

If the values in Table 1-1 represented microseconds, then a $O(\lg n)$ algorithm may take 20 microseconds to process 1,048,476 items, a $O(n^{1.25})$ algorithm might take 33 seconds, and a $O(n^2)$ algorithm might take up to 12 days! In the following chapters a timing estimate for each algorithm, using big-O notation, will be included. For a more formal derivation of these formulas you may wish to consult the references.

https://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_man.pd

Summary

- As we have seen, sorted arrays may be searched efficiently using a binary search.
- However, we must have a sorted array to start with.
- In what follows, various ways to sort arrays will be examined.
- It turns out that this is computationally expensive, and considerable research has been done to make sorting algorithms as efficient as possible.
- Linked lists improved the efficiency of insert and delete operations, but searches were sequential and time-consuming.
- Algorithms exist that do all three operations efficiently.

https://www.cs.auckland.ac.nz/~jmor159/PLDS210/niemann/s_man.pd

Question

Why isn't it a good idea to use binary search to find a value in a sorted **linked list** of values?

Question

Why isn't it a good idea to use binary search to find a value in a sorted **linked list** of values?

ANSWER: Binary search relies on being able to access the k^{th} item in a sequence of values in O(1) time. This is possible when the values are stored in an array, but not when they're stored in a linked list. So binary search using a linked list would usually be much slower than sequential search.

Definition

Sorting is the process of:

- Taking a list of objects which could be stored in a linear order

$(a_0, a_1, ..., a_{n-1})$

e.g., numbers, and returning a reordering

$(a'_0, a'_1, ..., a'_{n-1})$

such that

$a'_0 \leq a'_1 \leq \cdots \leq a'_{n-1}$

The conversion of an Abstract List into an Abstract Sorted List

Sorting Algorithms

- Consider sorting the values in an array A of size n.
- Most sorting algorithms involve what are called **comparison sorts**; i.e., they work by comparing values.
- Comparison sorts can never have a worst-case running time less than O(n log n).
- Simple comparison sorts are usually $O(n^2)$; the more clever ones are O(n log n).
- See https://codeworldtechnology.wordpress.com/2009/06/24/different-types-of-sorting-algorithms/

Types of Sorting Algorithms

- Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- Simple implementation: Jon Bentley shows a three-line C version, and a five-line optimized version[2]:116
- Efficient for (quite) small data sets, much like other quadratic sorting algorithms
- More efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(nk)$ when each element in the input is no more than k places away from its sorted position
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space
- Online; i.e., can sort a list as it receives it

Types of Sorting Algorithms

- Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

The idea behind insertion sort is:

1. Put the first 2 items in correct relative order.
2. Insert the 3rd item in the correct place relative to the first 2.
3. Insert the 4th item in the correct place relative to the first 3.
4. etc.

- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space
- Online; i.e., can sort a list as it receives it

Insertion Sort

Insertion sort is a brute-force sorting algorithm that is based on a simple method that people often use to arrange hands of playing cards: Consider the cards one at a time and insert each into its proper place among those already considered (keeping them sorted). The following code mimics this process in a Java method that sorts strings in an array:

```
public static void sort(String[] a) {
    int n = a.length;
    for (int i = 1; i < n; i++) {
        for (int j = i; j > 0; j--) {
            if (a[j-1].compareTo(a[j]) > 0)
                exch(a, j, j-1);
            else break;
        }
    }
}
```

http://introcs.cs.princeton.edu/java/42sort/

Types of Sorting Algorithms

- Insertion Sort
- Bubble Sort

Bubble sort, sometimes referred to as **sinking sort**, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. Passes through the list are repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort.[2] It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.

Types of Sorting Algorithms

- Insertion Sort
- Bubble Sort
- Heap Sort

heap

In certain programming languages including C and Pascal , a heap is an area of pre-reserved computer main storage (memory) that a program process can use to store data in some variable amount that won't be known until the program is running. For example, a program may accept different amounts of input from one or more users for processing and then do the processing on all the input data at once. Having a certain amount of heap storage already obtained from the operating system makes it easier for the process to manage storage and is generally faster than asking the operating system for storage every time it's needed. The process manages its allocated heap by requesting a "chunk" of the heap (called a *heap block*) when needed, returning the blocks when no longer needed, and doing occasional "garbage collecting," which makes blocks available that are no longer being used and also reorganizes the available space in the heap so that it isn't being wasted in small unused pieces.

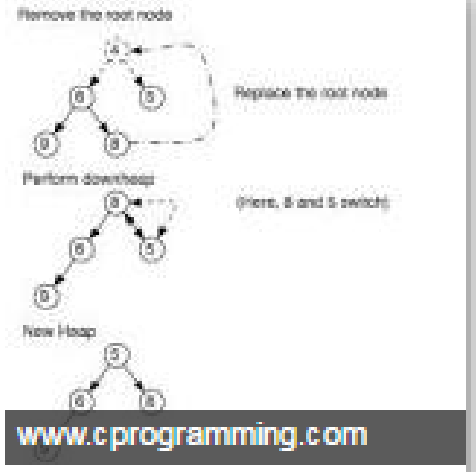
The term is apparently inspired by another term, stack . A stack is similar to a heap except that the blocks are taken out of storage in a certain order and returned in the same way. In Pascal, a *subheap* is a portion of a heap that is treated like a stack.

TechTarget | WhatIs.com

Types of Sorting Algorithms

- Insertion Sort
- Bubble Sort
- Heap Sort

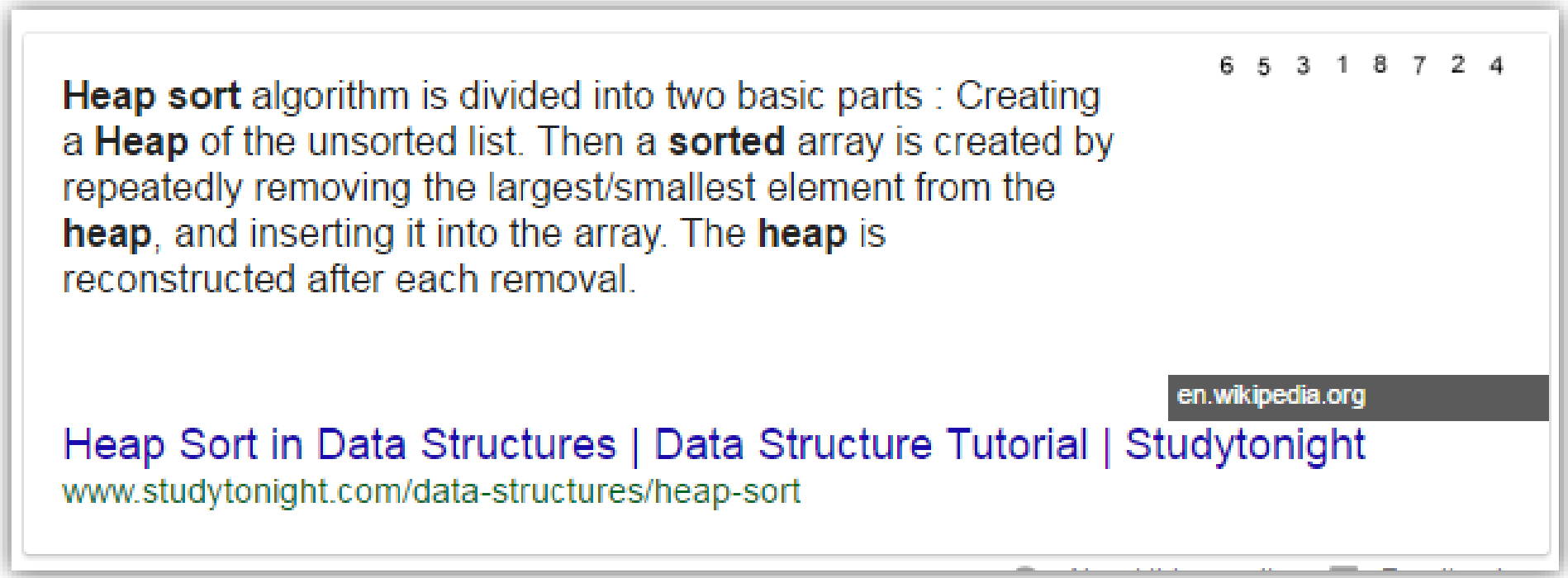
In computer science, a **heap** is a specialized tree-based **data structure** that satisfies the **heap** property: if P is a parent node of C, then the key (the value) of node P is ordered with respect to the key of node C with the same ordering applying across the **heap**.



Heap (data structure) - Wikipedia
https://en.wikipedia.org/wiki/Heap_(data_structure)

Types of Sorting Algorithms

- Insertion Sort
- Bubble Sort
- Heap Sort



Types of Sorting Algorithms

- Insertion Sort
- Bubble Sort
- Heap Sort
- Merge Sort

In computer science, **merge sort** (also commonly spelled **mergesort**) is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Mergesort is a divide and conquer algorithm that was invented by John von Neumann in 1945.[1] A detailed description and analysis of bottom-up mergesort appeared in a report by Goldstine and Neumann as early as 1948.[2]

END