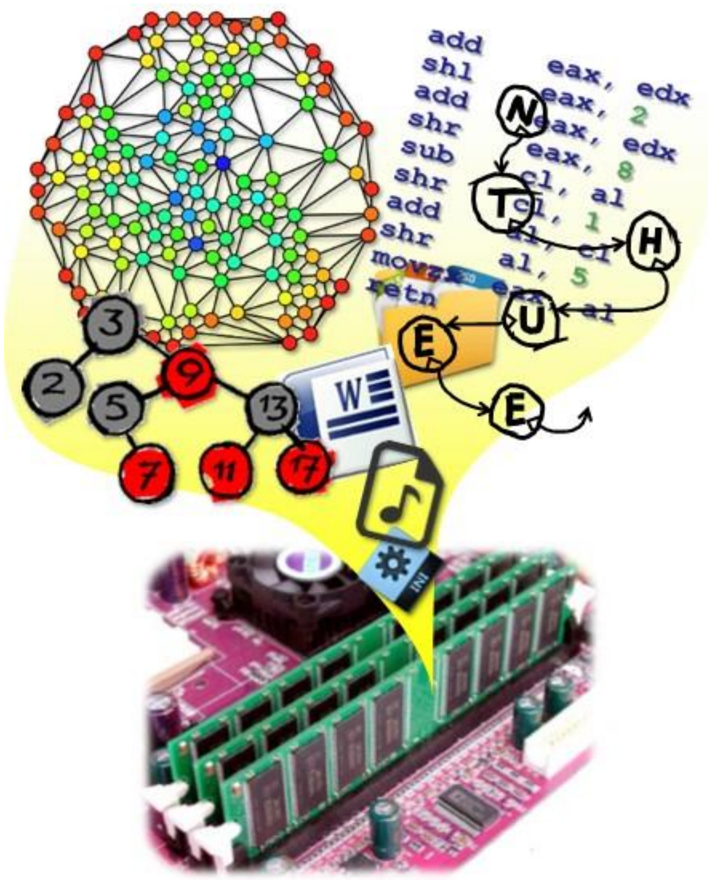
Data Structures

CH7 Sorting

Prof. Ren-Shuo Liu

NTHU EE

Spring 2018



Outline

- 7.1 Introduction
- 7.2 Insertion Sort
- 7.3 Quick Sort
- 7.4 How fast we can sort
- 7.5 Merge sort
- 7.6 Heap sort
- 7.7 Radix sort
- 7.8 (List and table sorts)
- 7.9 Summary of internal sorting

Important Uses of Sorting

- Aid searching in a list
  - O(n) time for an unordered list (with sequential search)
  - O(log(n)) time for a sorted list (with binary search)

- Aid matching two lists
  - O(nm) time for unordered lists (with sequential search)
  - O(n+m) time for sorted lists

Classification of Sorting

- Complexity
  - Time complexity
  - Space complexity
- Stability
  - A sort is called stable *iff* it maintains the relative order of records with equal keys
- Internal vs. external
  - An internal sort requires its inputs to be small enough so that the entire sort can be carried out in main memory
    - Examples: Selection Sort, Insertion Sort, Quick Sort, Heap Sort
  - An external sort has no abovementioned requirement

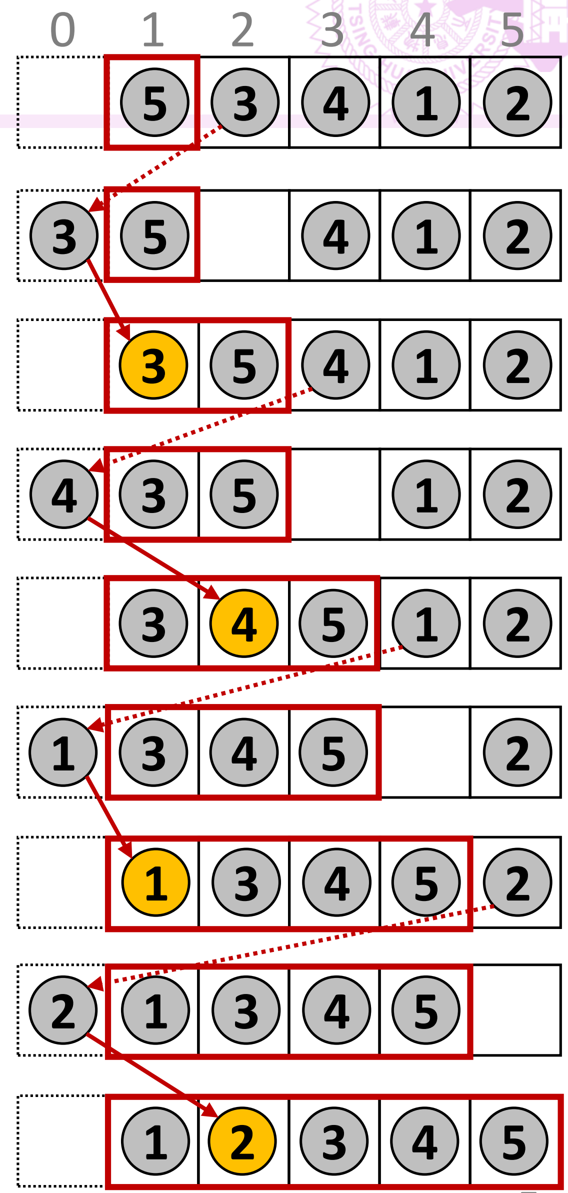
Score	Name
100	Alice
90	Bob
100	David
90	Emily

Score	Name
100	Alice
100	David
90	Emily
90	Bob

non-stable sort example

Insertion Sort Concept

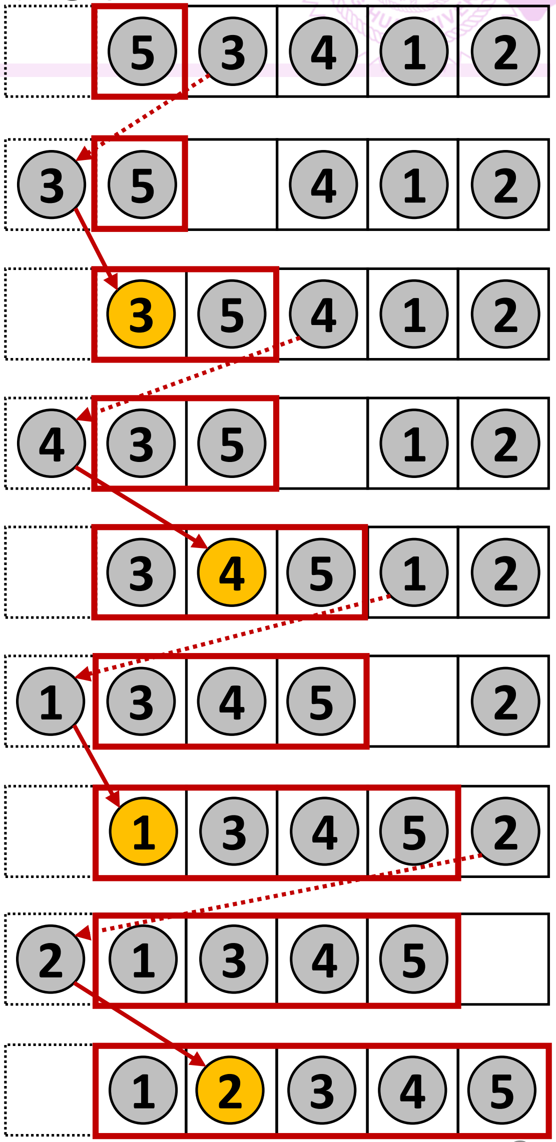
- array[0] is used as temporary space
- array[1] is the initial sorted sublist
- Insertion pass
  - Place the element next to the sublist to the temporary space
  - Insert the element to the sublist
- Insertion passes are continued until the sublist contains all records
- Insertion Sort is a stable sort



How Fast is Insertion Sort?

- Worst-case time complexity
  - When input is in a reversed order
  - Each insertion pass involves i comparisons, i = 1..n
  - 1+2+…+n = O(n^2)

- Average time complexity
  - It has been shown that Insertion Sort is O(n^2) on average



Insertion Sort Algorithm

```
template <class T>
void InsertionSort(T *a, const int n)
{ // sort a[1..n]
    for (int j = 2; j <= n ; j++){
        a[0] = a[j];
        for (int i = j - 1; a[i] > a[0]; i--) {
            a[i + 1] = a [i];
        }
        a[i + 1] = a[0];
    }
}
```

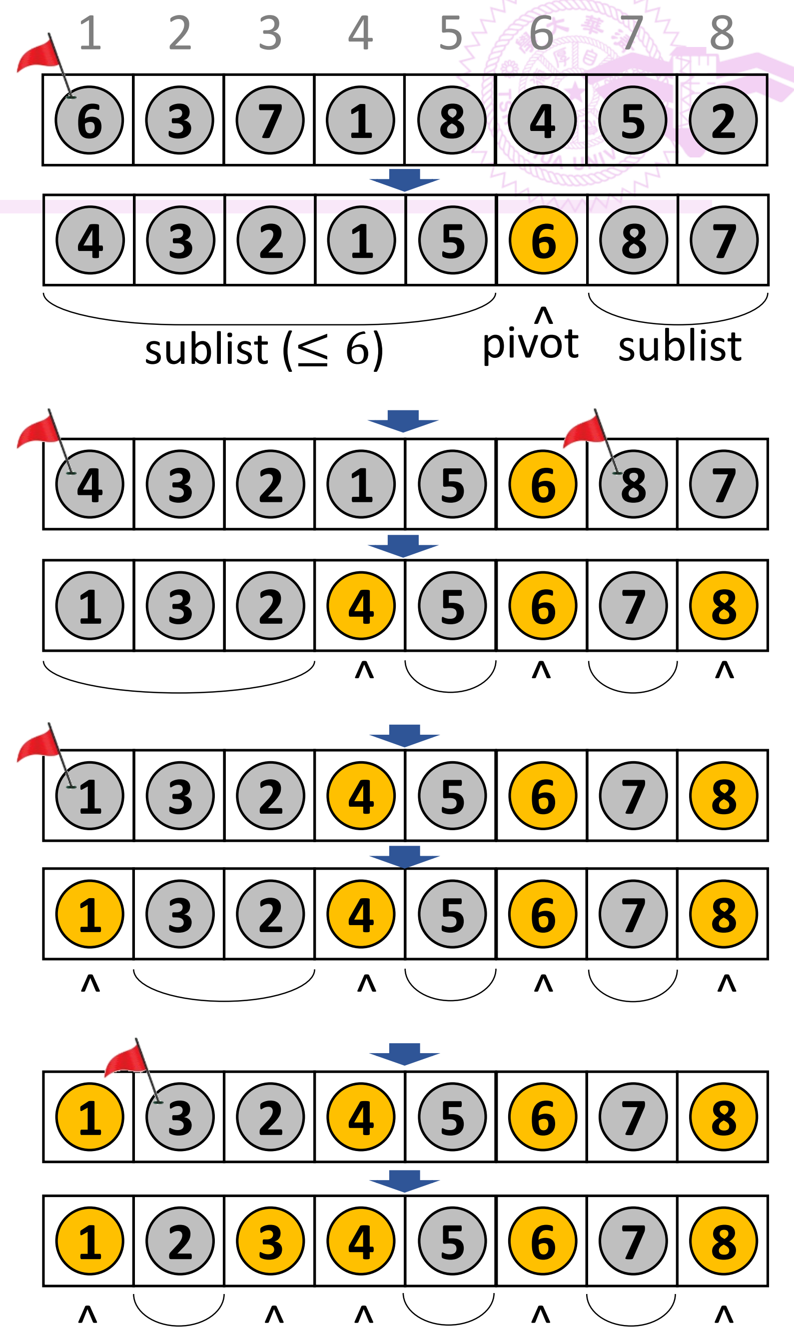
Variations of Insertion Sort

- Binary Insertion Sort
  - Use binary search rather than sequential search for insertion passes
  - Complexity does not change because the number of record moves remains unchanged

- Linked Insertion Sort
  - The records to be sorted are stored in a linked list rather than an array
  - The number of record moves becomes zero
  - Complexity does not change because sequential search is required for insertion

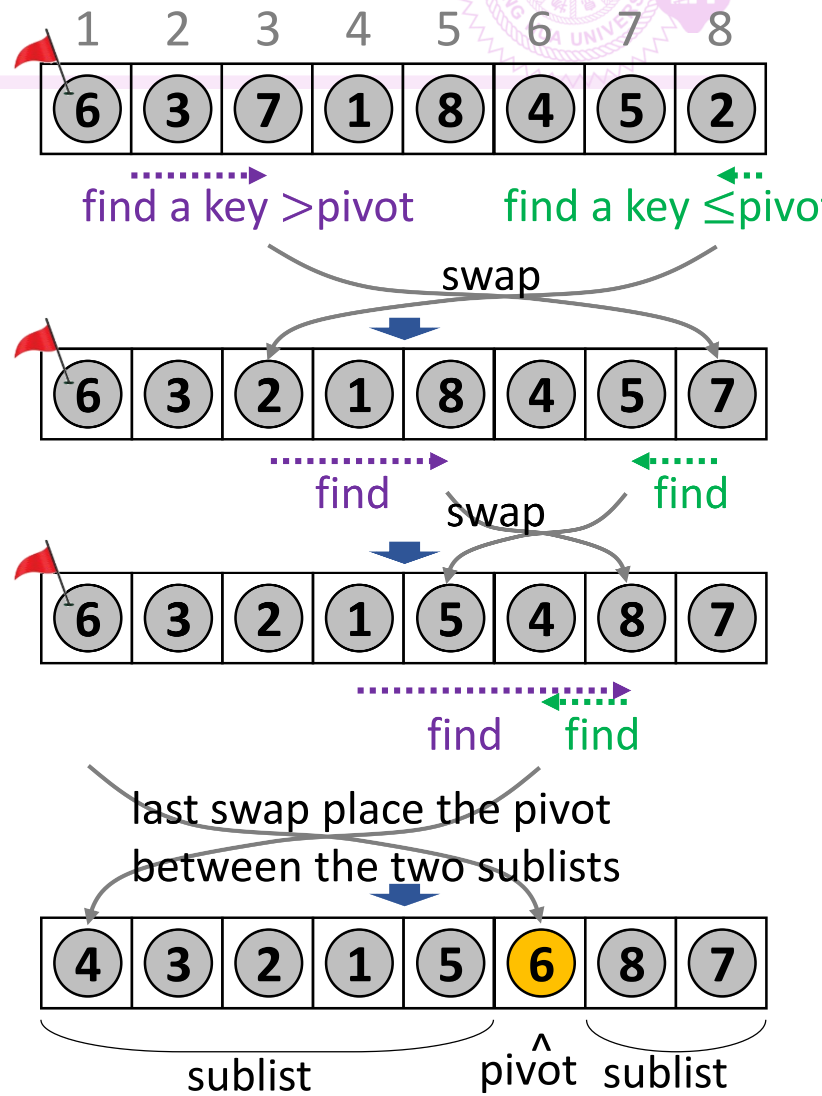
Quick Sort Concept

- Divide-and-conquer
- Division passes
    - Pick the first element of a list as the pivot
    - Make elements whose key ≤ the pivot be the left sublist
    - Make elements whose key > the pivot be the right sublist
- Division passes are continued until all sublists are of size ≤1
- Basically, Quick Sort is non-stable



Quick Sort Concept

- Steps for generating sublists
  - Linear searching from the both the ends
  - Find candidates to swap
  - Perform swapping



How Fast is Quick Sort?

- Worst-case time complexity
  - Each division pass involves n comparisons and end up with sublists with 1 and n-1 records
  - $T(n) = O(n) + T(n-1) = O(n^2)$
- Average time complexity
  - It has been shown that Quick Sort is $O(n \cdot \log(n))$ on average

Quick Sort Algorithm

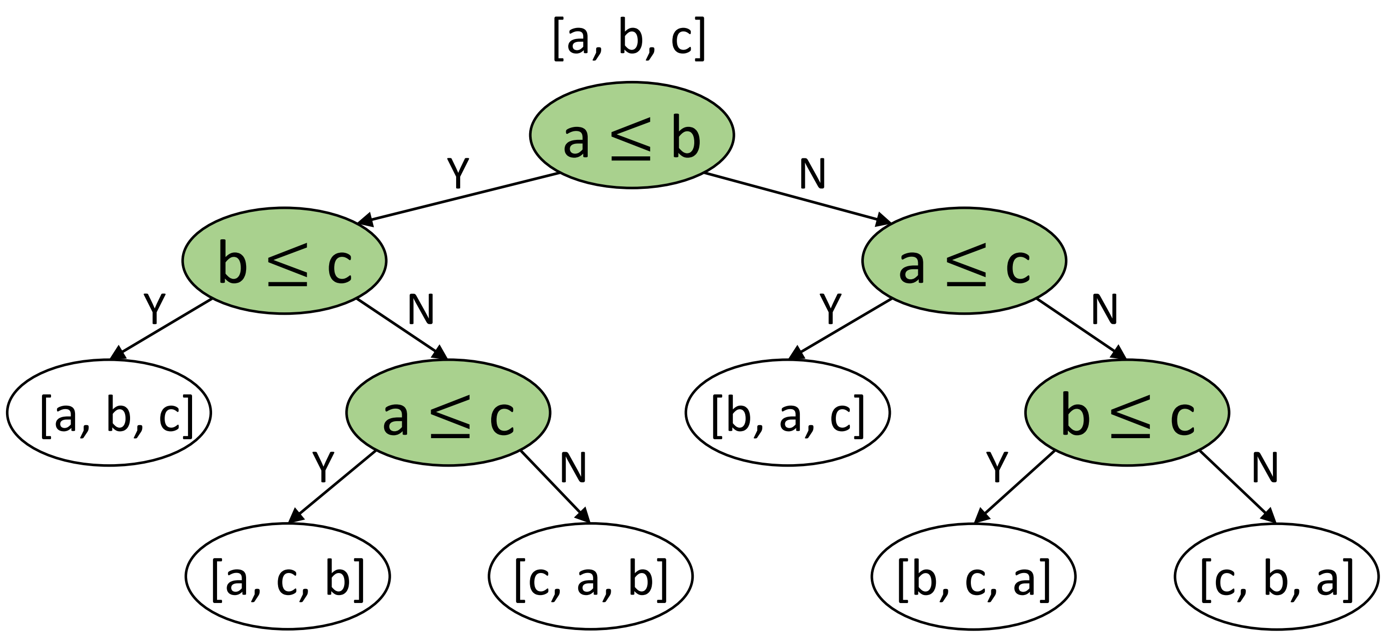
```
template <class T>
void QuickSort(T *a, const int left, const int right)
{ // sort a[left..right]
    if (left < right) {
        int & pivot = a[left];
        int i = left;
        int j = right + 1;
        do {
           do j--; while (a[j] > pivot);  //find a key ≤pivot
           do { i++;                      //find a key >pivot
           } while (i < j && a[i] <= pivot);
           if (i < j) swap (a[i], a[j]);
        } while (i < j);
        swap (pivot, a[j]); //place the pivot between 2 lists
        QuickSort(a, left, j - 1);  // recursion
        QuickSort(a, j + 1, right); // recursion
    }
}
```

Variations of Quick Sort

- Median-of-three strategy
  - Ordered lists are worst-case inputs for Quick Sort
    - Pivot are always the smallest or largest key within a sublist
    - Ordered lists are not rare in real life
  - Choosing the pivot using the median of the first, middle, and last key can address this issue

How Fast Can We Sort?

- A sorting algorithm can be represented as a binary decision tree
  - Non-leaf node represents a comparison between two keys
  - Leaf nodes are the sorting results

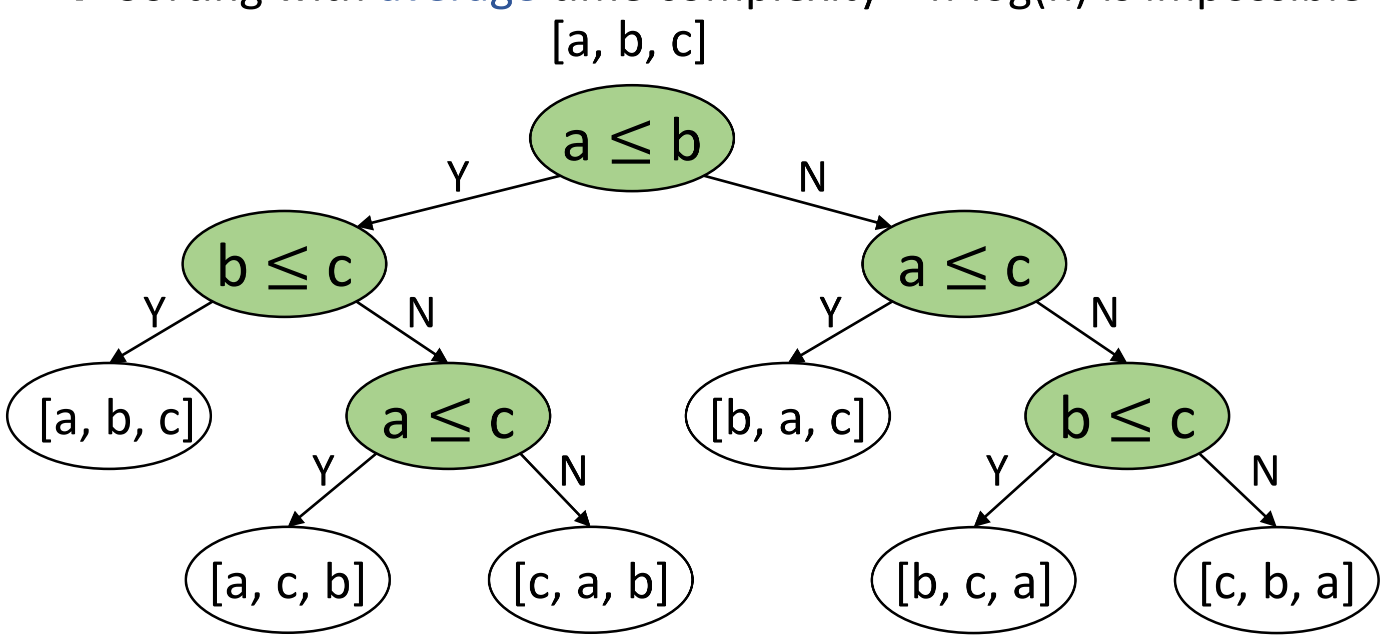


How Fast Can We Sort?

- Sorting n records
  - Number of leaf nodes is at least n!
  - Tree height is at least ($\log_2(n!) + 1$) = $\Omega(n \cdot \log(n))$

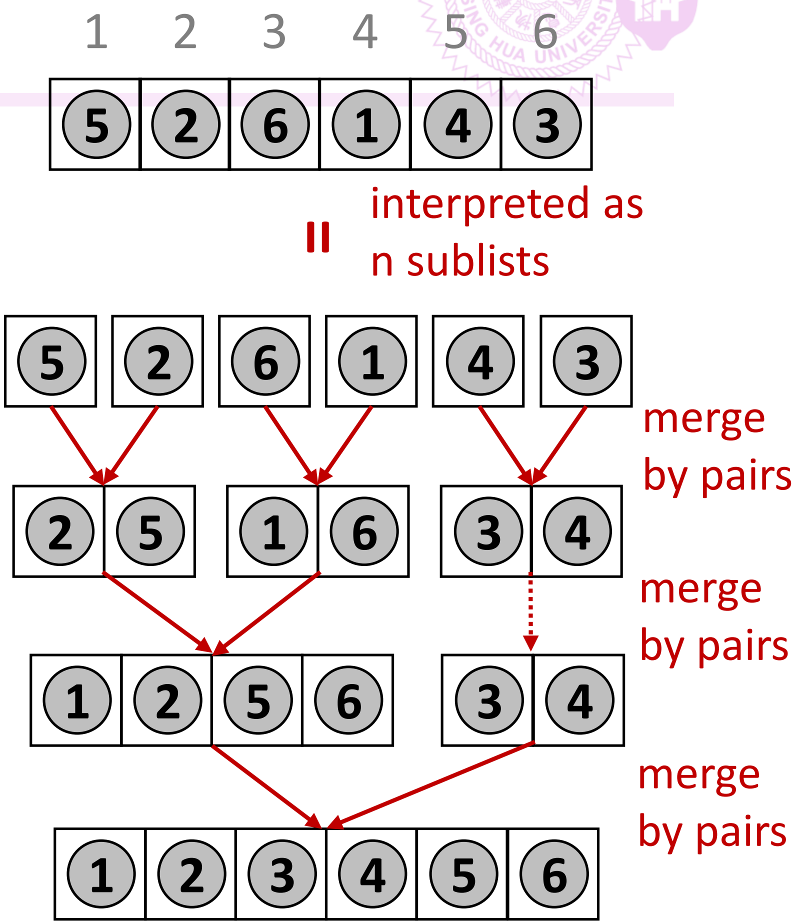
    → Sorting with worst time complexity < n·log(n) is impossible
  - Average root-to-leaf path length is $\Omega(n \cdot \log(n))$

    → Sorting with average time complexity < n·log(n) is impossible



Merge Sort

- Interpret the unsorted list as n sorted sublists, each of size 1
- These lists are merged by pairs in each pass
- Merge passes are continued until there is only one sublist remained
- Merge Sort is stable



How Fast is Merge Sort ?

- Both worst and average cases
  - $\log(n)$ merge passes are performed
  - Each merge pass is $O(n)$
  - Time complexity is $O(n \cdot \log(n))$

Merge Sort Algorithm

```
template <class T>
void MergeSort(T *a, const int n)
{ // sort a[1:n] into non-decreasing order
    T *tempList = new T[n+1];

    // s is the length of the currently merged sublist
    for (int s = 1; s < n; s *= 2)
    {
        MergePass(a, tempList, n, s);

        s *= 2;
        MergePass(tempList, a, n, s);
    }
    delete [] tempList;
}
```

(to be continued)

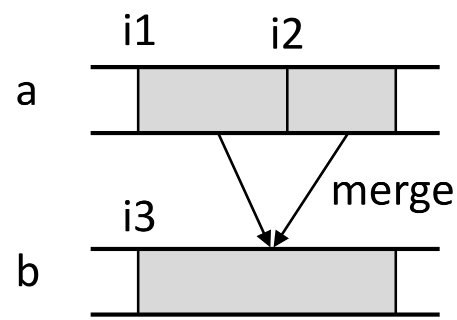
Merge Sort Algorithm

```
template <class T>
void MergePass(T *a, T *b, const int n, const int s)
{
    for (int i = 1;
          i <= n-(2*s)+1;
          i+ = 2*s) {
          Merge(a, b, i, i+s-1, i+(2*s)-1);
    }

    // merge remaining lists
    if ((i + s-1) < n)  // one full and one partial lists
        Merge(a, b, i, i+s-1, n);
    else                // only one partial lists remained
        copy(a+i, b+n+1, b+i);
}
```

(to be continued)

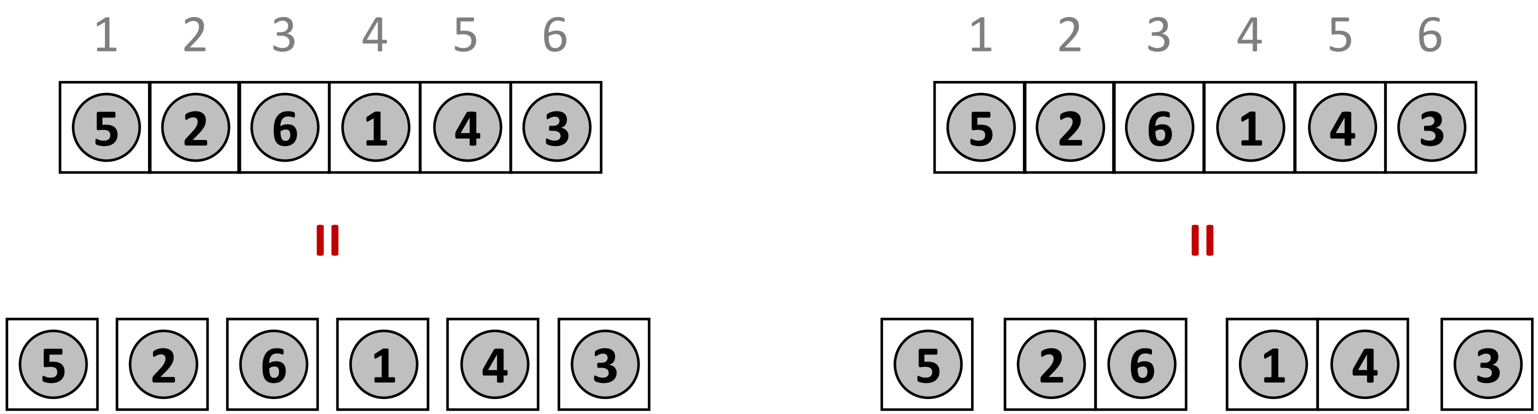
Merge Sort Algorithm

```
template <class T>
void Merge(T *a, T *b, const int k, const int m, const int n)
{
  for (int i1 = k, i2 = m+1, i3 = k;
        i1 <= m && i2 <= n;
        i3++) {
          if (a[i1] <= a [i2]) {
               b [i3] = a [i1];
               i1++;
          } else {
               b [i3] = a [i2];
               i2++;
          }
        // copy remaining records, if any, of 1st sublist
        copy (a+i1, a+m+1, b+i3);

        // copy remaining records, if any, of 2nd sublist
        copy (a+i2, a+n+1, b+i3);
  }
}
```



Variations of Merge Sort

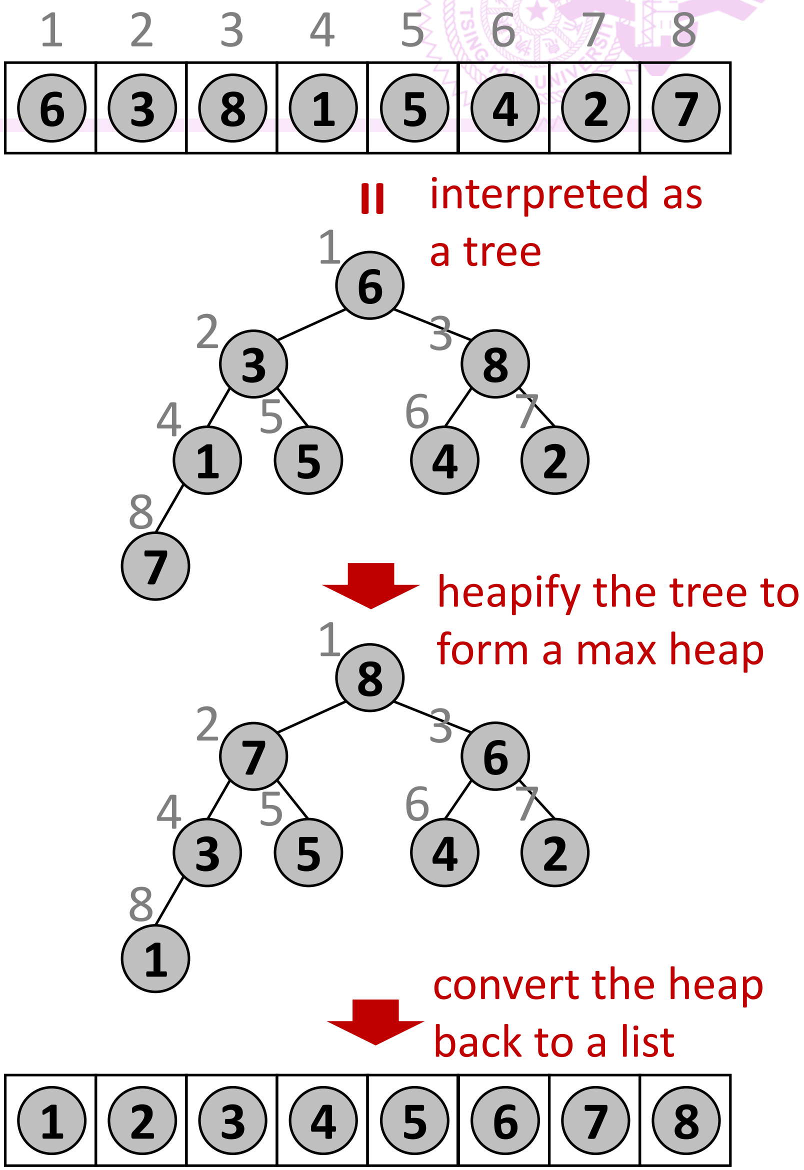
- Natural Merge Sort
  - Interpreting the initial list as multiple sorted sublists, each can contain more than one records



Original Implementation **Natural Merge Sort**

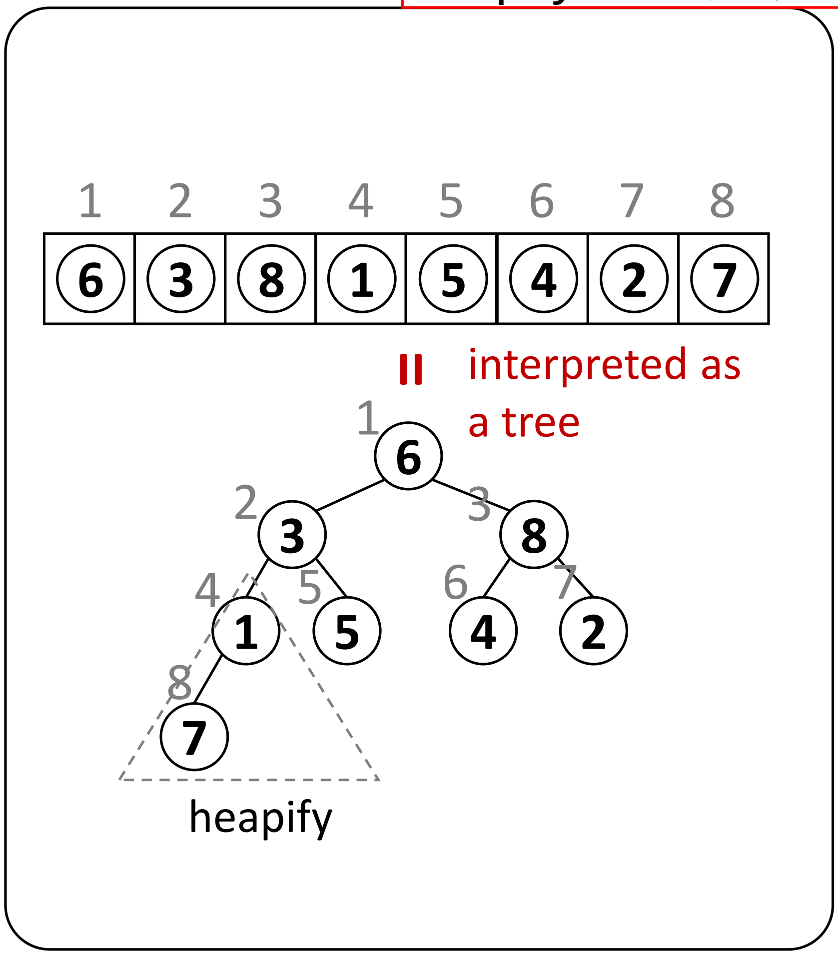
Heap Sort Concept

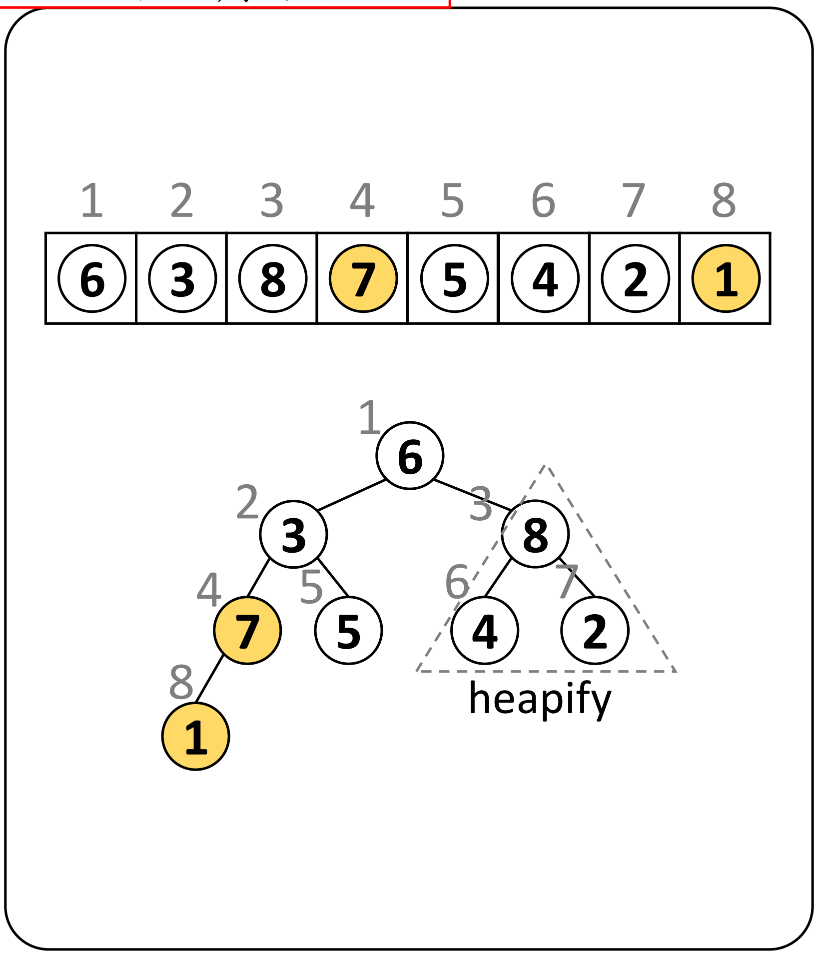
- Interpret the input list as a tree
- Heapify the tree to form a max heap
- Popping pass
  - Pop the top (maximum) record
    - Heap size shrinks by one
    - Space next to the heap becomes unused
  - Place the popped record at the space
- Popping passes are continued until the heap becomes empty
- Heap Sort is non-stable



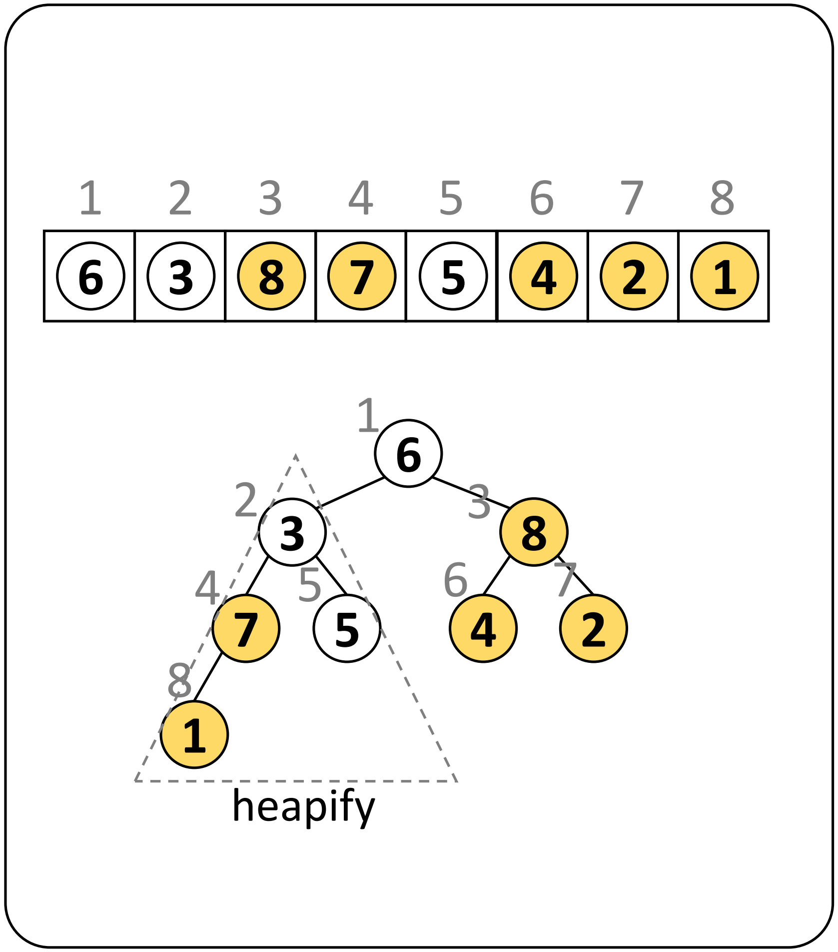
Heap Sort Detail Steps

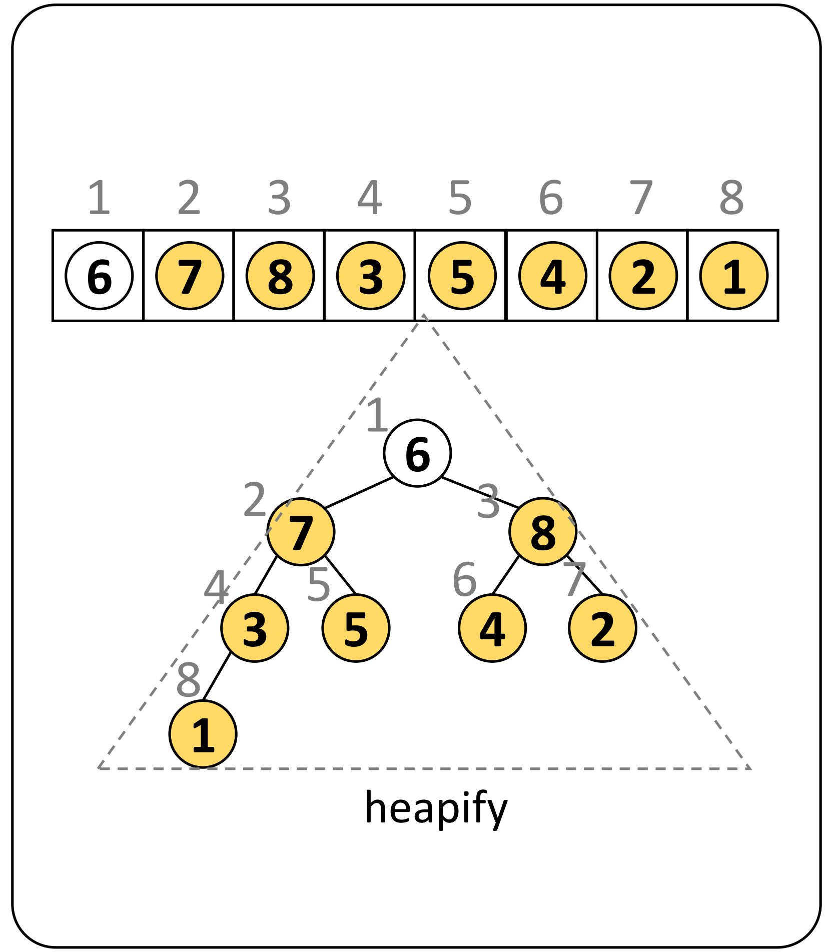
heapify：從小的三角形-->大的三角形檢查



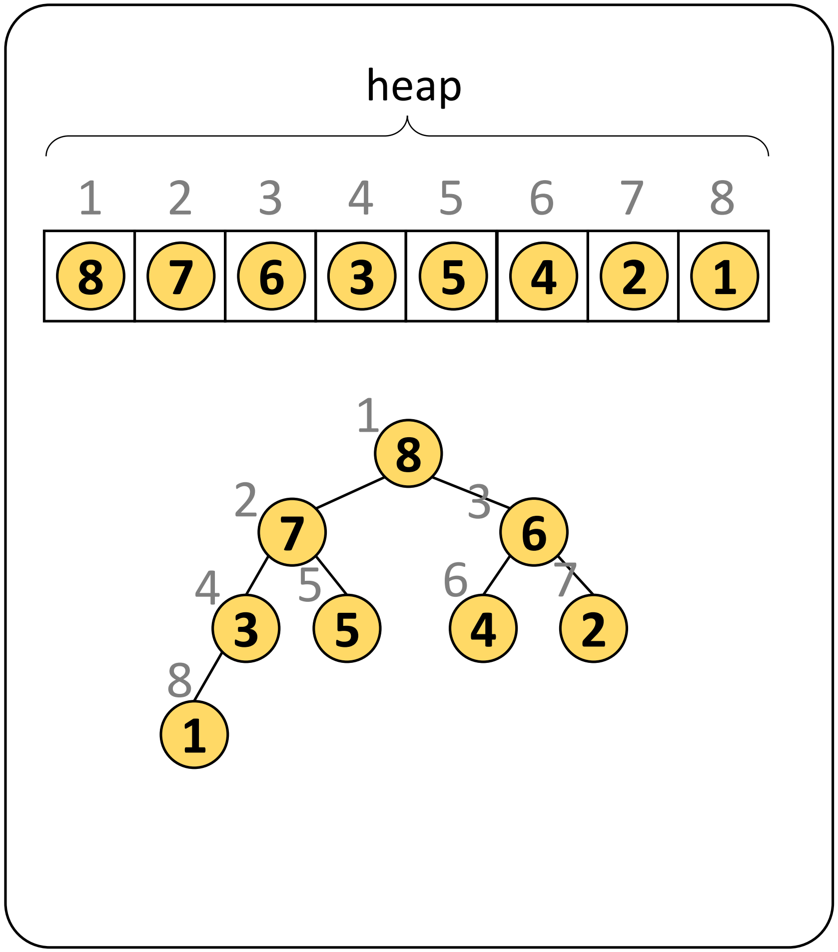


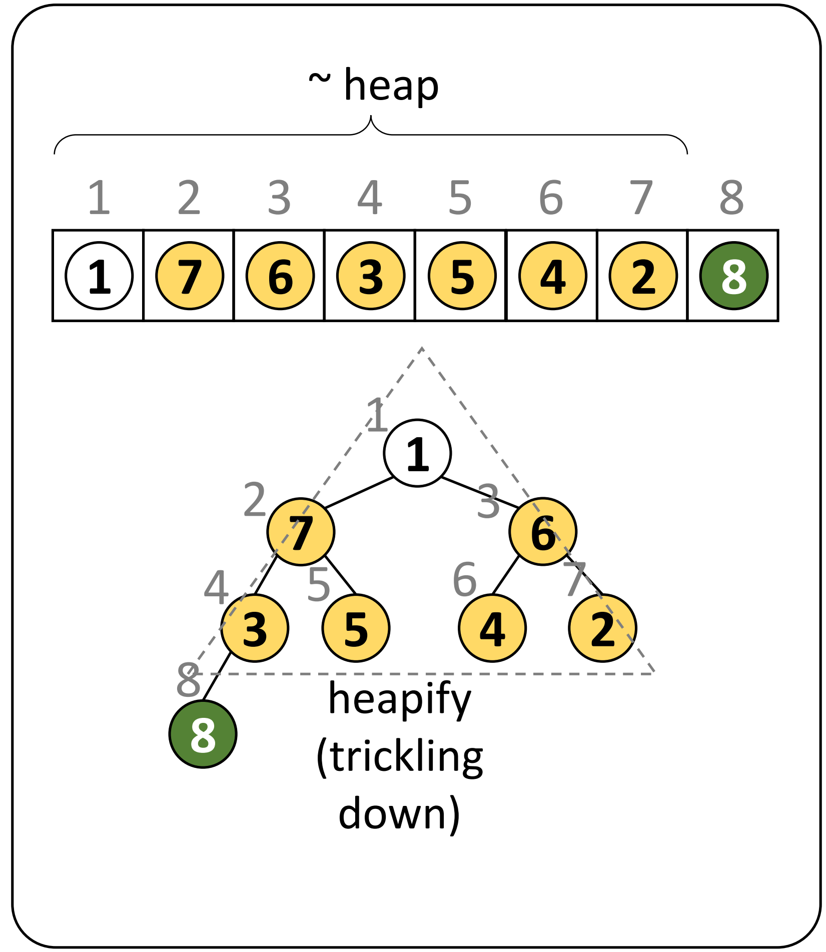
Heap Sort Detail Steps



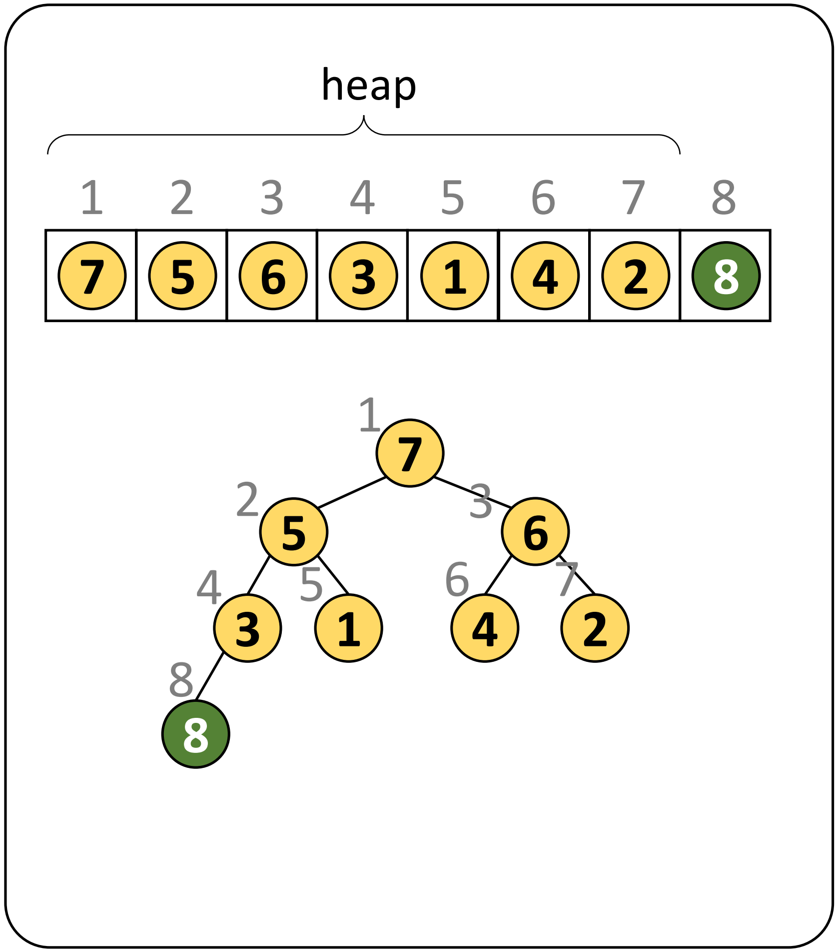


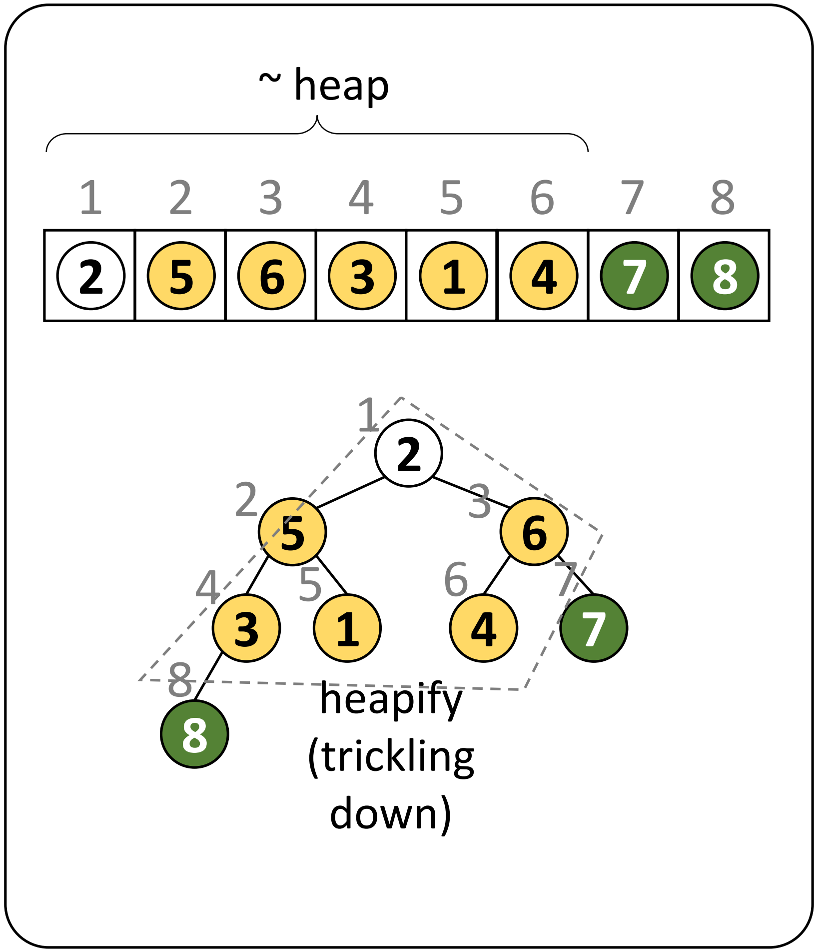
Heap Sort Detail Steps



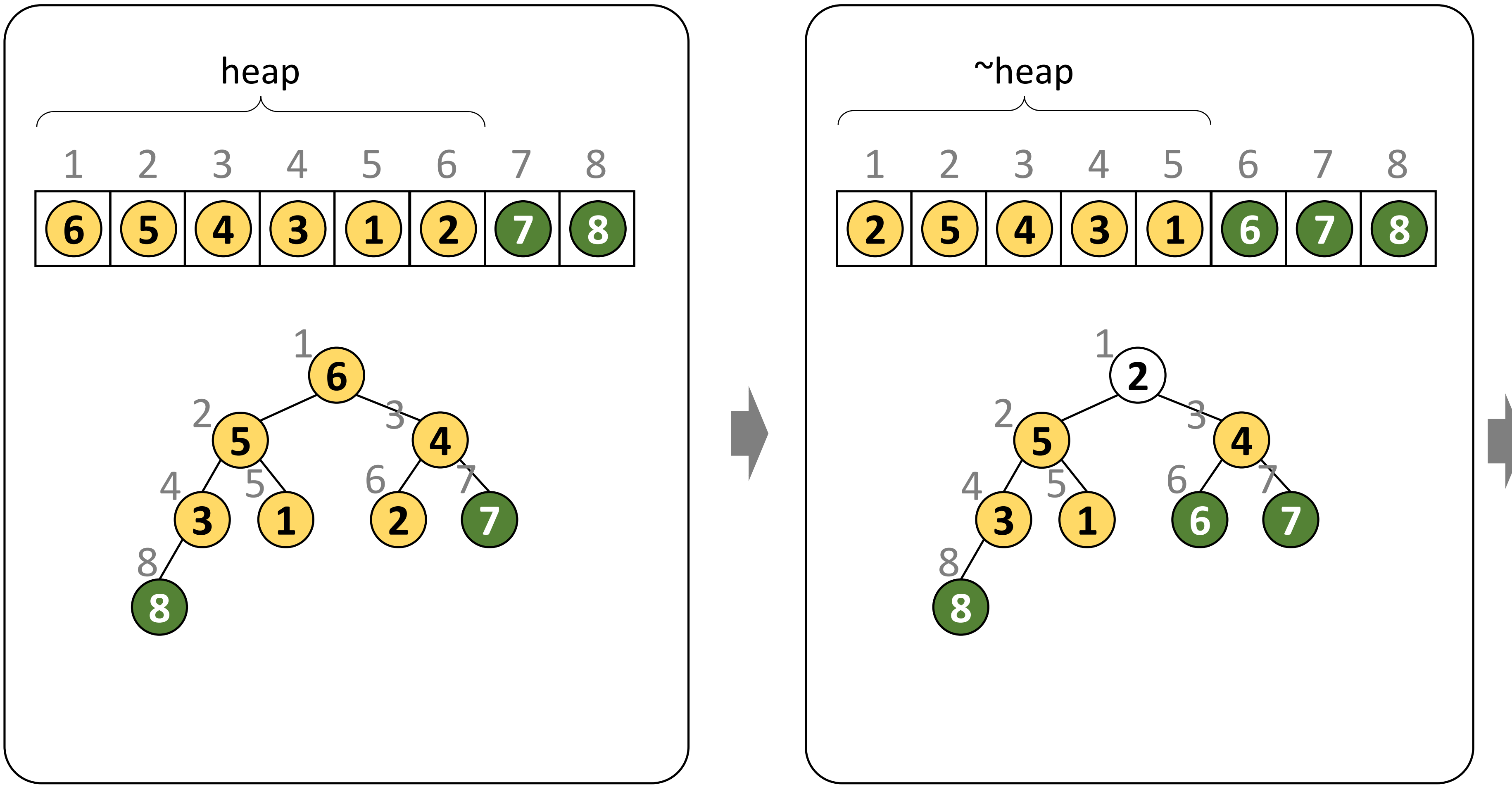


Heap Sort Detail Steps





Heap Sort Detail Steps



Heap Sort Algorithm

```
template <class T>
void HeapSort(T *a, const int n)
{// sort a[1..n] into non-decreasing order
     // a[n/2] is the parent of the last node, a[n]
    for (int i = n/2; i >= 1; i--) // buttom-up heapification
        Adjust(a, i, n);  // make the tree rooted at i be a max heap

    for (int i = n-1; i >= 1; i--) {
        swap(a[1], a[i+1]); // move one record from heap to list
        Adjust(a, 1, i);    // heapify
    }
}
```

(to be continued)

Heap Sort Algorithm

```
template <class T>
void Adjust(T *a, const int root, const int n)
{
    // two subtrees are max heaps already
    // same procedure as the trickling-down procedure
    T e = a[root];
    //2*root is root's left child
    for (int j = 2*root; j <= n; j *=2) {
        if (j < n && a[j] < a[j+1])  // j and j+1 are siblings
            j++; // make j be the max child
        if (e >= a[j])
            break;
        a[j / 2] = a[j]; // move jth record up the path
    }
    a[j / 2] = e;
}
```

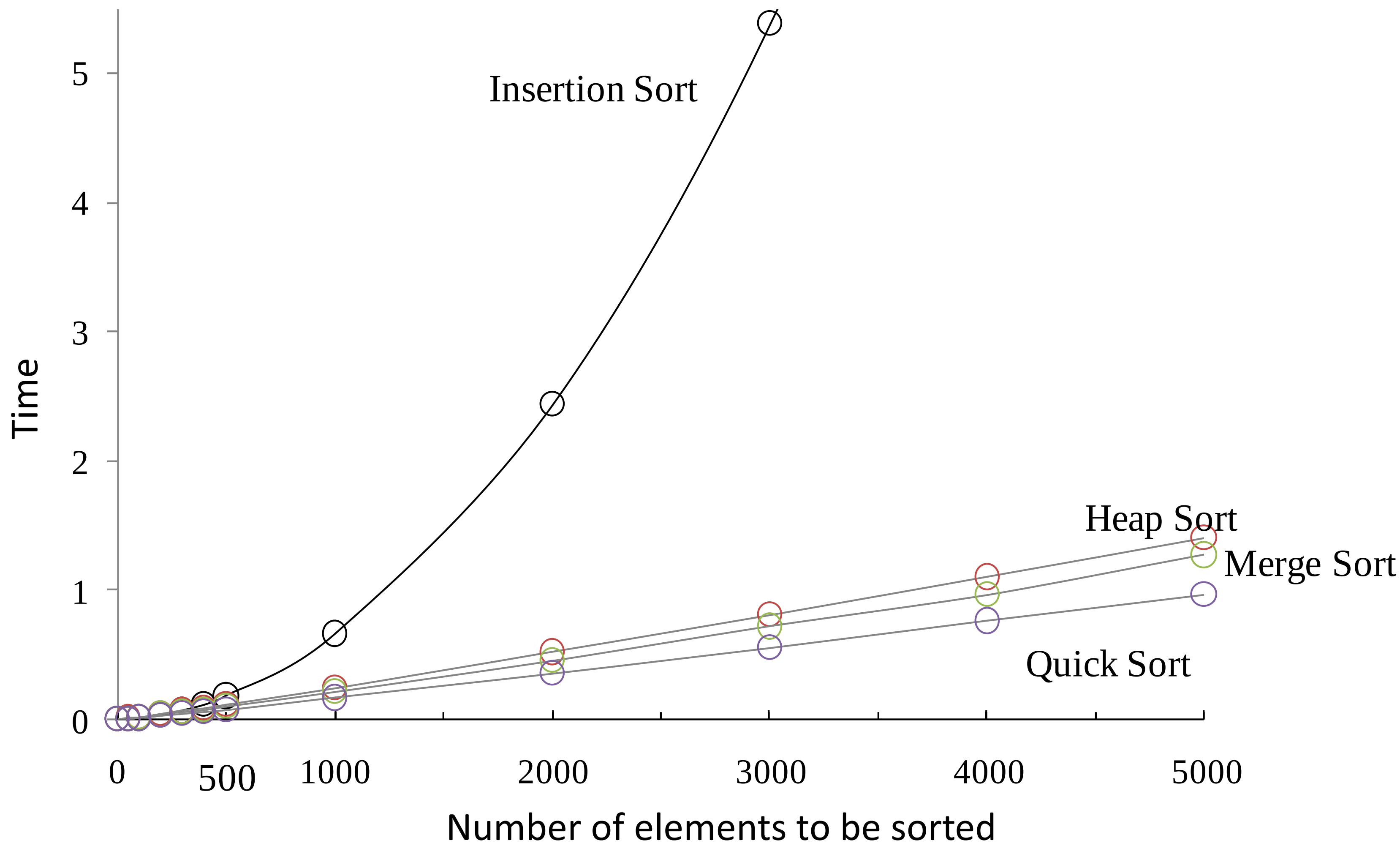
How Fast is Heap Sort?

- Both worst and average cases
  - Heapifying the tree
    - n/2 adjust()'s are invoked, each is at most O(log(n))
  - Converting the max heap to the list
    - n pop()'s are invoked, each is O(log(n))
  - Overall, the time complexity is O(n·log(n))

Summary

	Worst	Average	
Insertion Sort	n^2	n^2	• Fastest method when n is small (e.g., n<100) • O(1) space • Stable
Quick Sort	n^2	nlogn	• Fastest method in practice • Require $O(n^2)$ time in the worst case • Require O(log(n)) space • Non-stable
Merge Sort	n·log(n)	n·log(n)	• Require additional O(n) space • Stable
Heap Sort	n·log(n)	n·log(n)	• Require additional O(1) space • Non-stable

Summary



Time
5
4
3
2
1
0
Insertion Sort
Heap Sort
Merge Sort
Quick Sort
0
500
1000
2000
3000
4000
5000
Number of elements to be sorted

Outline

- 7.1 Introduction
- 7.2 Insertion Sort
- 7.3 Quick Sort
- 7.4 How fast we can sort
- 7.5 Merge sort
- 7.6 Heap sort
- **7.7 Radix sort**
- 7.8 List and table sorts
- 7.9 Summary of internal sorting

Sorting on Several Keys

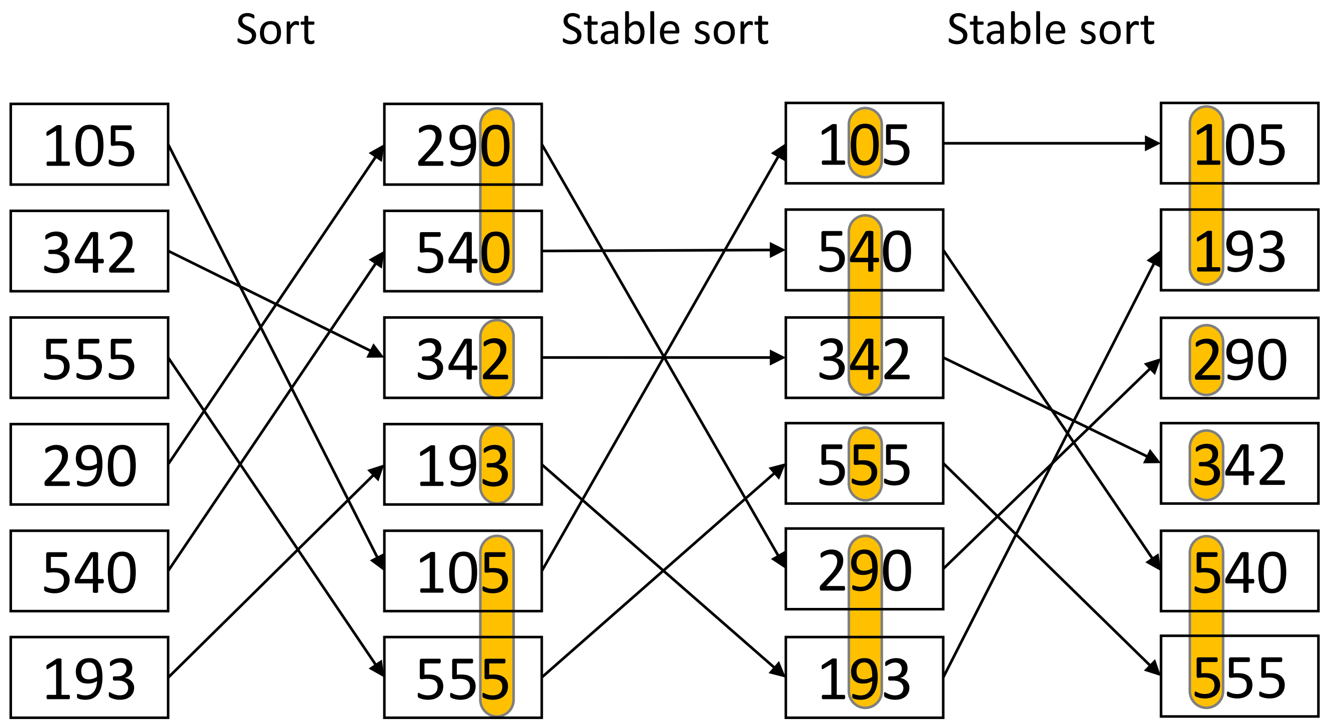
- Sorting a deck of cards
  - Sort on two keys
    - Suits (most-significant digit, MSD) : ♣ < ♦ < ♥ < ♠
    - Face values (least-significant digit, LSD) : 2 < 3 <… < Q < K < A

- Two popular sorting strategies
  - MSD first sort
  - LSD first sort

Radix Sort

- Decompose each key into several keys using some radix
  - e.g., 365 is decomposed into 3, 6, and 5 with a radix = 10
- Common practices
  - LSD-first sort is commonly chosen for computer sorting
  - MSD-first sort tends to incur much overhead because of the need to independently sort multiple groups

LSD-First Radix Sort Example



Summary

- Every sorting algorithm has its pros and cons
  - No one size fit all solution
- C++'s sort methods
  - sort()
    - Quick Sort that reverts to Heap Sort when the recursion depth exceeds some threshold and to Insertion Sort when the segment size becomes small
  - stable_sort()
    - Merge Sort that revers to Insertion Sort when the segment size becomes small
  - partial_sort()
    - Heap Sort that has ability to stop when only the first k elements need to be sorted