University of Kelaniya Sri Lanka

COSC 21063

Bubble Sort & Heap Sort

Lecturer : Ms.D.M.L.Maheshika Dissanayake

Structure of Lesson

- Bubble Sort/ Exchange sort
  - Definition
  - Method
  - Implementation
  - Analysis
- Heap Sort
  - Definition
  - Method
  - Implementation
  - Analysis

Learning Outcomes

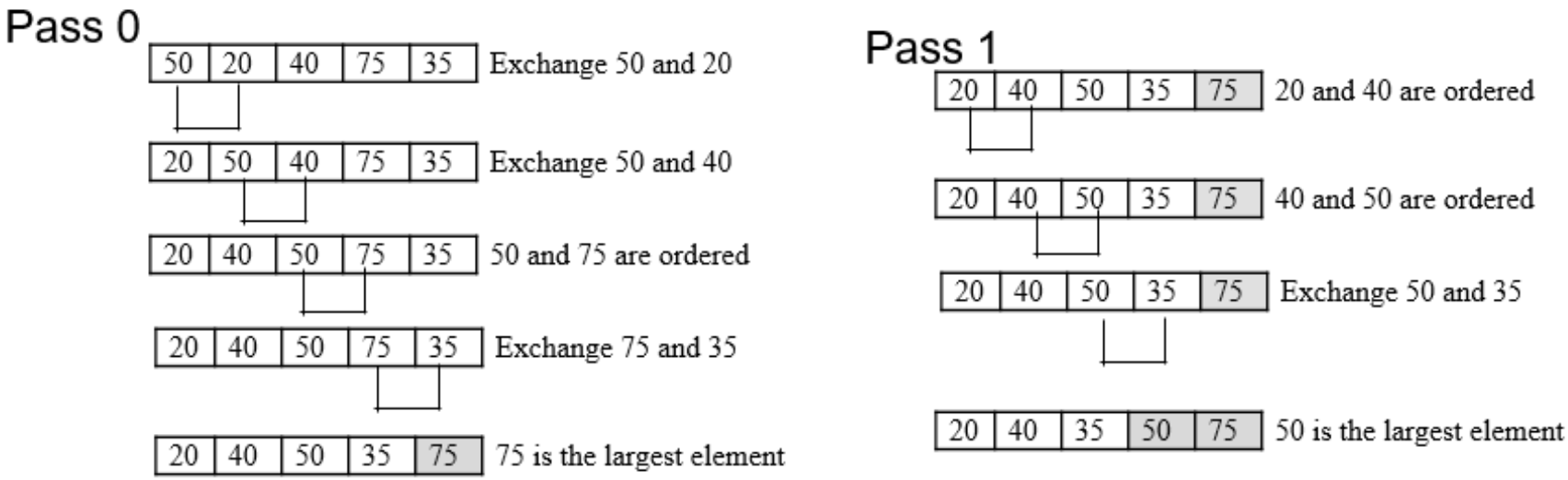
By the end of this lesson you should be able to

- Understand the concept of bubble/ exchange sort and heap sort
- Implement the bubble sort and heap sort algorithms in java language
- Analyze the algorithms

Bubble Sort/ Exchange sort

- Proceeds by scanning the list from left to right, swapping entries whenever a pair of adjacent keys is found to be out of order.
- We will consider the bubble sort for contiguous list.
- Bubble sort requires up to n –1 passes. For each pass, we compare adjacent elements and exchange their values when the first element is greater than the second.
- At the end of each pass, the largest element has “bubbled up” to the top of the current list.
- For example, after the first pass (pass 0) the tail of the list i.e. list[n-1] is sorted and the front of the list remains unordered.

Example : Consider a list with elements 50, 20, 40, 75,35



Implement the bubble sort for the contiguous list.

```
void bubbleSort() {
    int n = ListEntry.length;
    int temp = 0;
    for(int i=0; i < n; i++){
        for(int j=1; j < (n-i); j++){
            if(ListEntry[j-1] > ListEntry[j]){
                //swap elements
                temp = ListEntry[j-1];
                ListEntry[j-1] = ListEntry[j];
                ListEntry[j] = temp;
            }
        }
    }
}
```

Analysis

- The inner loop is executed the following number of times.

  $$(n-1) + (n-2) + \ldots\ldots + (n-i_{sorted}) = (i_{sorted}(2n - i_{sorted} - 1))/2 \quad // \; n*i - i(i+1)/2$$

- where i_{sorted} is the number of executions of the outer loop.
- The inner loop contains one comparison and sometimes one exchange.

Analysis

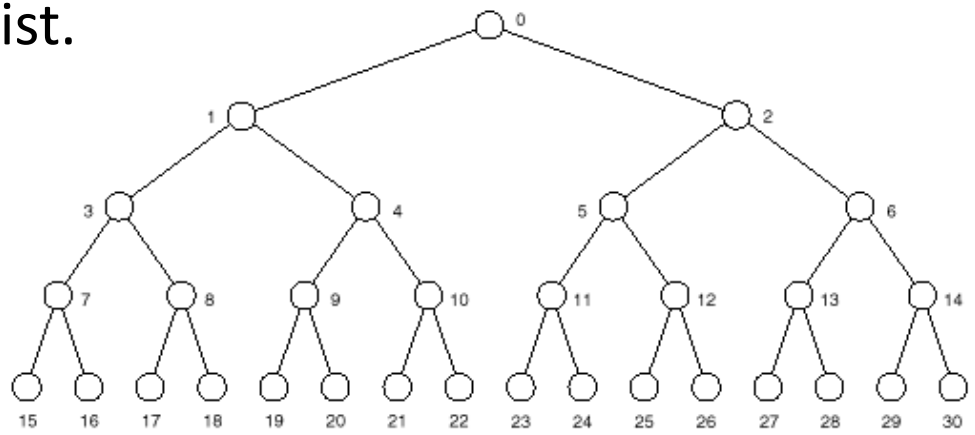
- The best performance occurs when i_{sorted} is 1. This means no exchanges occur because the original data was sorted.
- In the best case the number of comparison is 1 therefore

  $(2n-2)/2 = (n-1) = n + O(1)$
- the number of exchanges = 0
- The worst performance occur when i_{sorted} is n-1 (the list is in descending order)
- The inner loop is executed the following number of times

  $((n-1)*n)/2 = n2/2 + O(n)$ (for both number of comparison and number of exchanges)

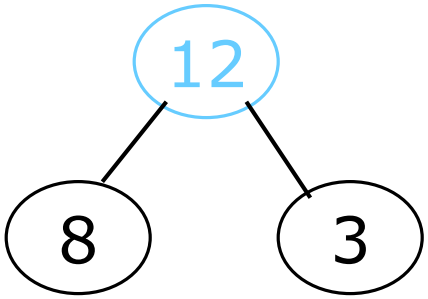
Heap Sort

- Heapsort is based on a tree structure that reflects the pecking order in a corporate hierarchy.
- Two-way trees as Lists
- The left and right children of the node with position *k* are in positions 2*k+1* and 2*k* +2 of the list, respectively.
- If these positions are beyond the end of the list, then these children do not exist.

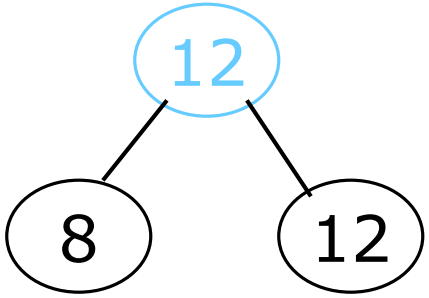


The heap property

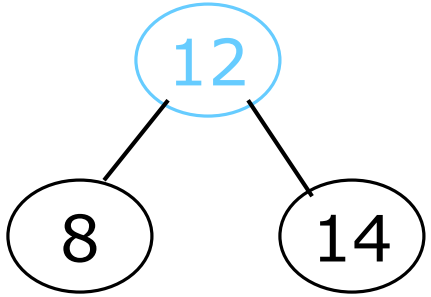
- A node has the heap property if the value in the node is as large as or larger than the values in its children



Blue node has heap property



Blue node has heap property

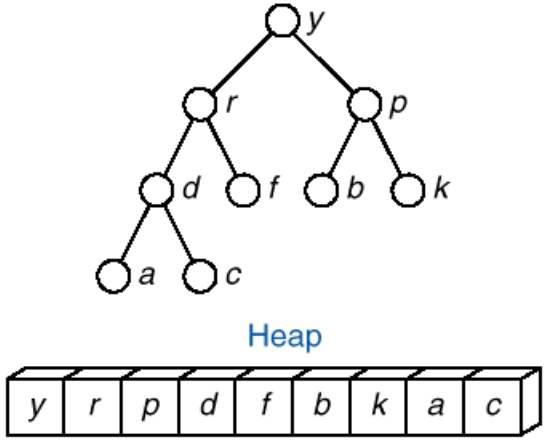


Blue node does not have heap property

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

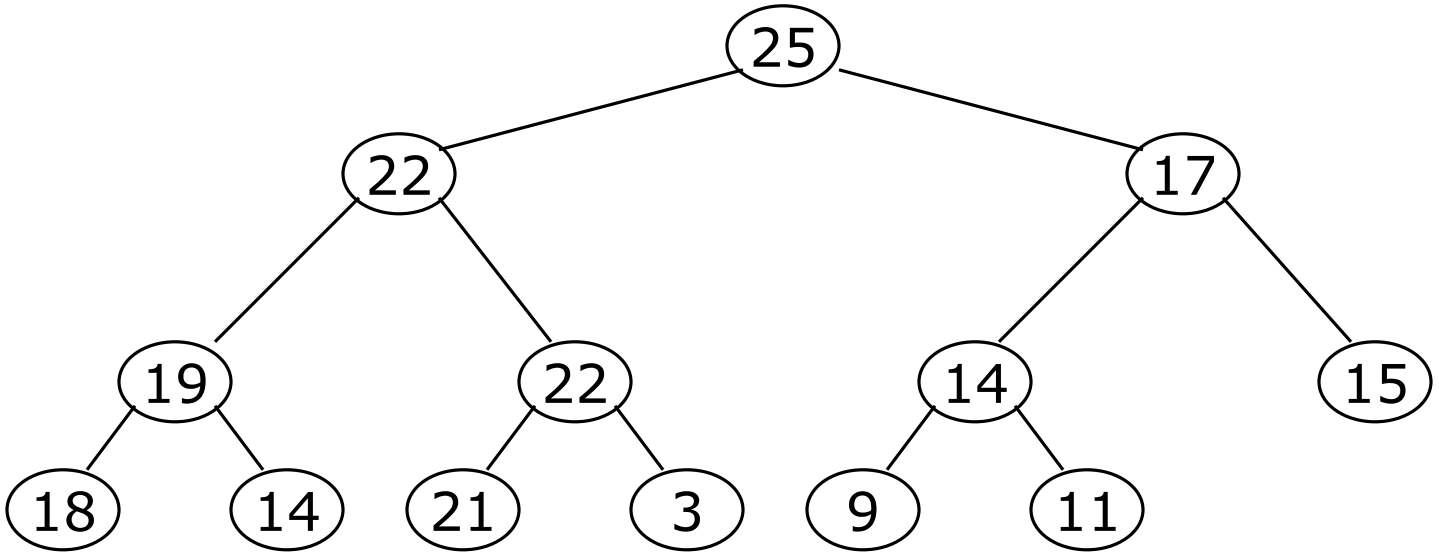
Heap Sort

- A *heap* is a list in which each entry contains a key, and, for all positions *k* in the list, the key at position *k* is at least as large as the keys in positions 2*k+1* and 2*k +2*, provided these positions exist in the list.
- We draw trees as shown - A heap as a tree and as a list



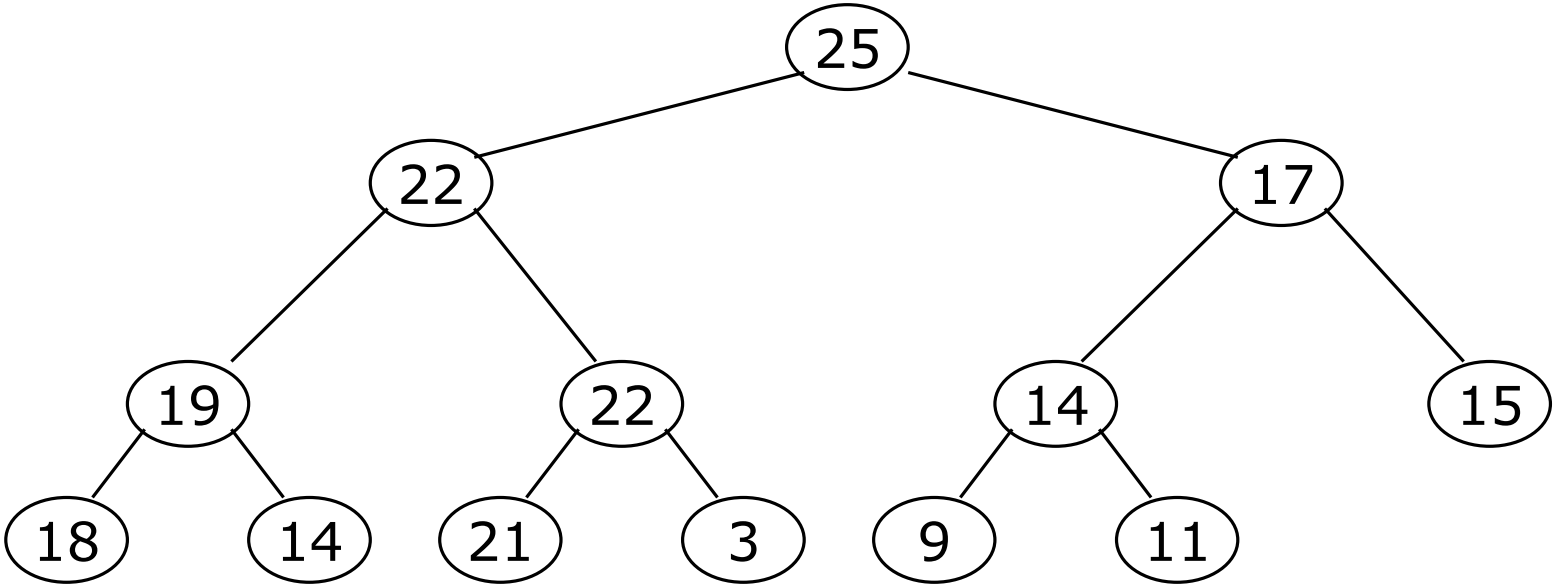
- Heap is not an ordered list.
- The first entry must have the largest key in the heap.
- There is no necessary ordering between the keys in locations k and k +1 if k >0

A sample heap



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

Mapping into an array



0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

Notice:

- The left child of index k is at index 2k+1
- The right child of index k is at index 2k+2
- Example: the children of node 3 (19) are 7 (18) and 8 (14)

Method

Heapsort proceeds in two phases.

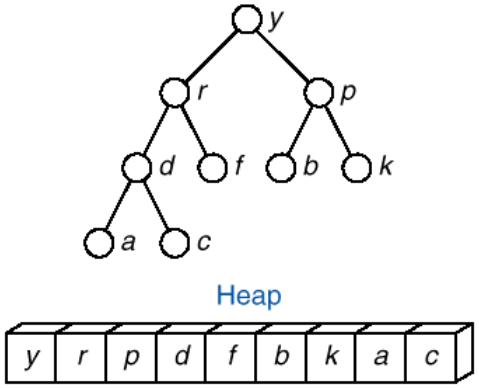
1. We must arrange the entries in the list so that they satisfy the requirements for a heap.
2. We repeatedly remove the top of the heap and promote another entry to take its position.

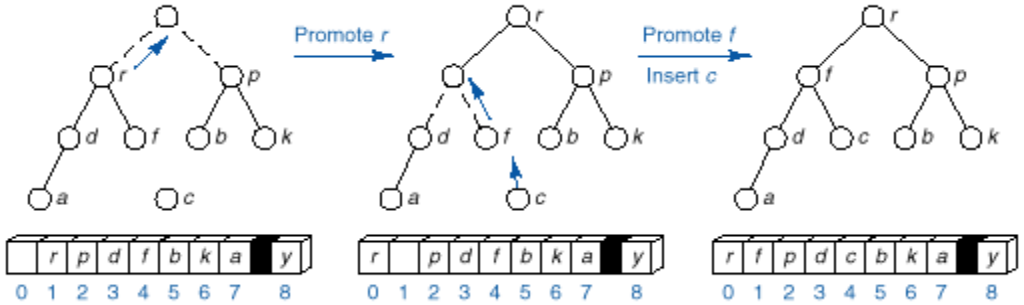
For 2,

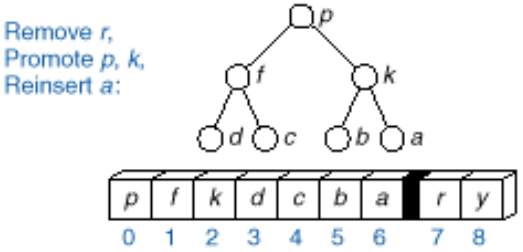
- the root of the tree which is the first entry of the list as well as the top of the heap) has the largest key.
- This key belongs to the end of the list.
- Move the first entry to the last position, replacing an entry *current*. (i.e. the last entry in the list will be *current*)

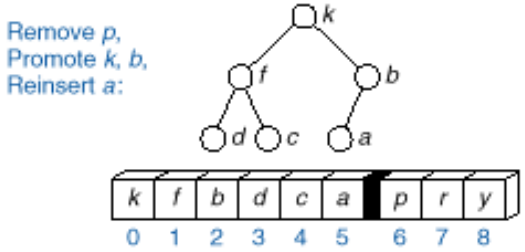
Method

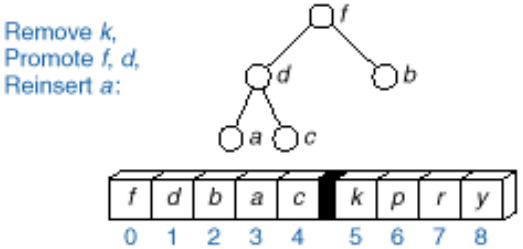
- We then decrease a counter that keeps track of the size of the unsorted part of the list, thereby excluding the largest entry from further sorting.
- The entry *current* that has been removed from the last position may not belong to the top of the heap, and we must insert *current* into the proper position to restore the heap properly.
- Then continue the loop in the same way.
- Heapsort requires random access to all parts of the list.
- Heapsort is only suitable for contiguous lists.

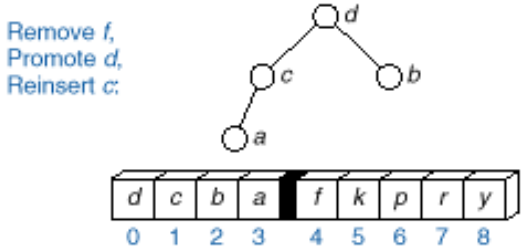


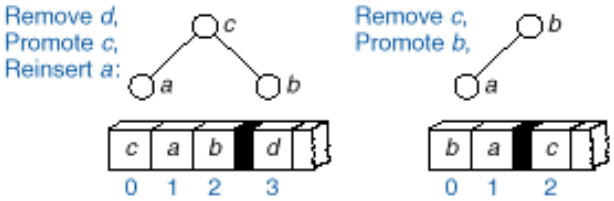












- The largest entry *y* is moved from the first to the last position.
- *c* is put as the temporary variable current.

```
void heapSort()
   {
      int n = ListEntry.length;
// build the heap
      for (int i = n / 2 - 1; i >= 0; i--)
         InsertHeap(n, i);

      // One by one extract an element from heap
      for (int i = n - 1; i >= 0; i--) {
         /* Move current root element to end*/
         // swap a[0] with a[i]
         int temp = ListEntry[0];
         ListEntry[0] = ListEntry[i];
         ListEntry[i] = temp;
         InsertHeap(i, 0);
      }
   }
```

```
void InsertHeap(int n, int i) {
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && ListEntry[left] > ListEntry[largest])
        largest = left;
    // If right child is larger than root
    if (right < n && ListEntry[right] > ListEntry[largest])
        largest = right;
    // If root is not largest
    if (largest != i) {
        // swap a[i] with a[largest]
        int temp = ListEntry[i];
        ListEntry[i] = ListEntry[largest];
        ListEntry[largest] = temp;
        InsertHeap( n, largest); }
  }
```

Analysis

Heapsort is not suitable for short lists

Move large keys slowly

For contiguous lists guaranteed to finish in time O(nlog(n)), with minimal space requirements.

Summary of main points

- Definition of Bubble sort and Heap Sort
- Implementation of Bubble sort and Heap sort
- Analysis of bubble and heap sort algorithms