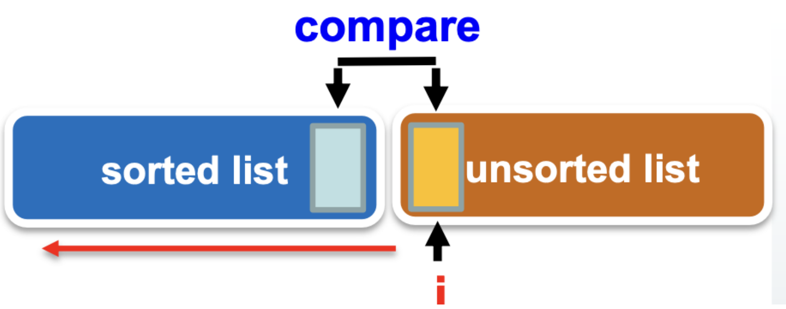
Sorting notes

Algorithm Design and Analysis (Nanyang Technological University)

Sorting Algorithms

Insertion sort



Pseudocode

```
for (int i=1; i < n; i++)
        for (int j=i; j > 0; j--) {
                if (slot[j].key < slot[j-1].key)
                        swap(slot[j], slot[j-1]);
                else break;
        }
```

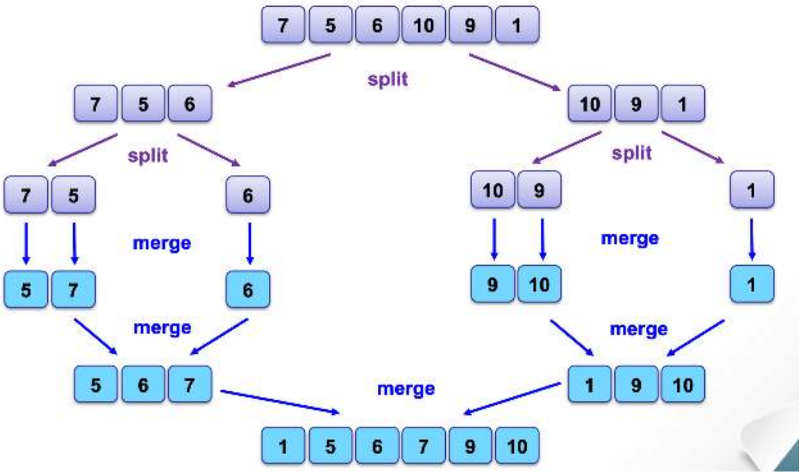
Complexity

Number of key comparisons: There are n−1 iterations (the outer loop)

Best Case (input array already sorted)	• Every loop only requires one comparison • Total no. of comparisons: n-1 • Time complexity: O(n)
Worst Case (input array reversely sorted)	• Every loop requires the maximum number of comparisons • Total no. of comparisons: $$1+2+3+...+(n-1)=\sum_{i=1}^{n-1} i=\frac{(n-1)n}{2}$$ • Time complexity: $O(n^2)$
Average Case	• The ith iteration may have 1,2,...,i key comparisons, each with 1/i chance • Total no. of comparisons $$\frac{1}{i}\sum_{j=1}^{i} j=\frac{1}{i}(1+2+...+i)$$

	$1+\frac{1}{2}(1+2)+\frac{1}{3}(1+2+3)+...+\frac{1}{n-1}(1+...+n-1) = \sum_{i=1}^{n-1}(\frac{1}{i}\sum_{j=1}^{i} j)$ $= \sum_{i=1}^{n-1}\left(\frac{1}{i}\frac{i(i+1)}{2}\right) = \frac{1}{2}\sum_{i=1}^{n-1}(i+1) = \frac{1}{2}\left(\frac{(n-1)(n+2)}{2}\right) = \Theta(n^2)$ ● Time complexity: $O(n^2)$

Merge sort



Pseudocode

<table>
<tr>
<td>MergeSort</td>
<td>

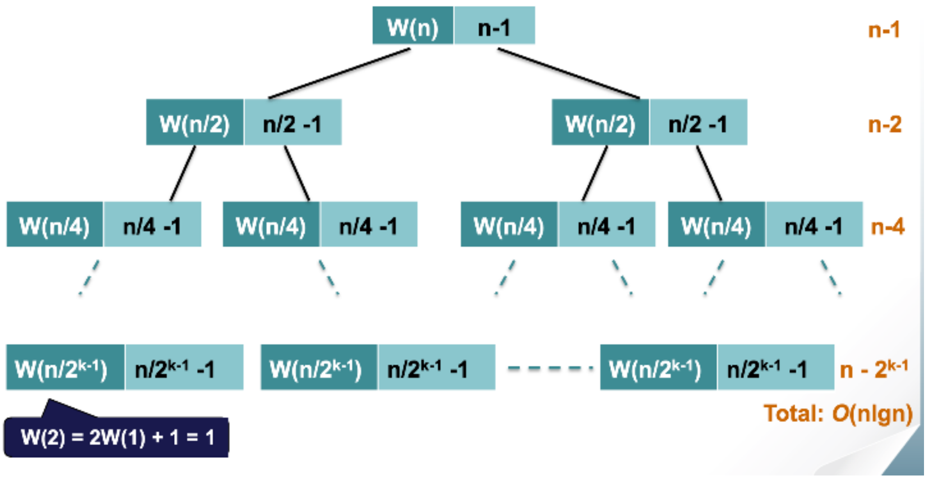
```
void mergesort(int n, int m)
{   int mid = (n+m)/2;
    if (m-n <= 0)
        return;
    else if (m-n > 1) {
        mergesort(n, mid);
        mergesort(mid+1, m);
    }
    merge(n, m);
}
```

</td>
</tr>
</table>

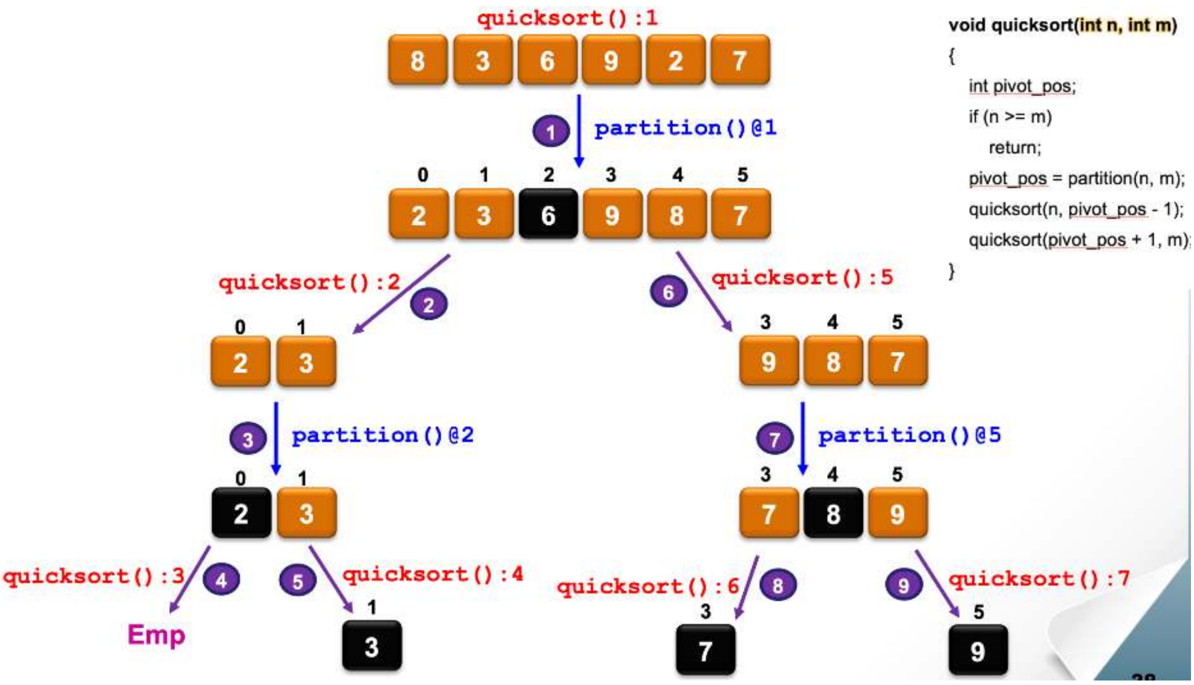
<table>
<tr><td>Merge</td><td>void merge(int n, int m) {
if (m-n <= 0) return;
divide the list into 2 halves; // both halves are sorted
while (both halves are not empty) {
compare the 1st elements of the 2 halves; // 1 comparison
if (1st element of 1st half is smaller)
1st element of 1st half joins the end of the merged list;
else if (1st element of 2nd half is smaller)
move the 1st element of 2nd half to the end of the merged list;
else { // the 1st elements of the 2 halves are equal
if (they are the last elements) break;
1st element of 1st half joins end of the merged list;
move the 1st element of 2nd half to the end of the merged list;
}
} // end of while loop;
} // end of merge</td></tr>
</table>

Complexity

<table>
<tr><td>Merge()</td><td>• All cases have same number of key comparisons: n-1</td></tr>
<tr><td>MergeSort()</td><td>Worst case :
$W(1) = 0,$
$W(n) = W(n/2) + W(n/2) + n-1$ Or
$W(2^k) = 2W(2^{k-1}) + 2^k - 1$
$= 2(2W(2^{k-2}) + 2^{k-1} - 1) + 2^k - 1$
$= 2^2W(2^{k-2}) + 2^k - 2 + 2^k - 1$
$= 2^2(2W(2^{k-3}) + 2^{k-2} - 1) + 2^k - 2 + 2^k - 1$
$= 2^3W(2^{k-3}) + 2^k - 2^2 + 2^k - 2 + 2^k - 1$
…
$= 2^kW(2^{k-k}) + k2^k - (1 + 2 + 4 + \dots + 2^{k-1})$
$= k2^k - (2^k - 1)$
$= n \lg n - (n - 1)$
$= O(n \lg n)$</td></tr>
</table>

	 • Height of tree: lgn • Each level of tree $W(n/2^{k-1})$ has $n-2^{k-1}$ comparisons • Total no. of key comparisons = O(nlgn)

Quick sort

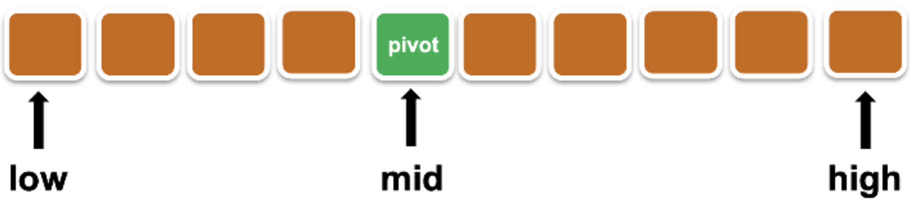


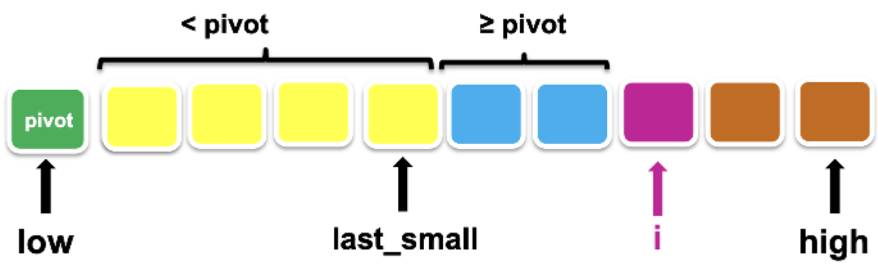
Pseudocode

QuickSort

```
void quicksort(int n, int m)
{
    int pivot_pos;
    if (n >= m)
        return;
    pivot_pos = partition(n, m);
    quicksort(n, pivot_pos - 1);
    quicksort(pivot_pos + 1, m);
}
```

Partition





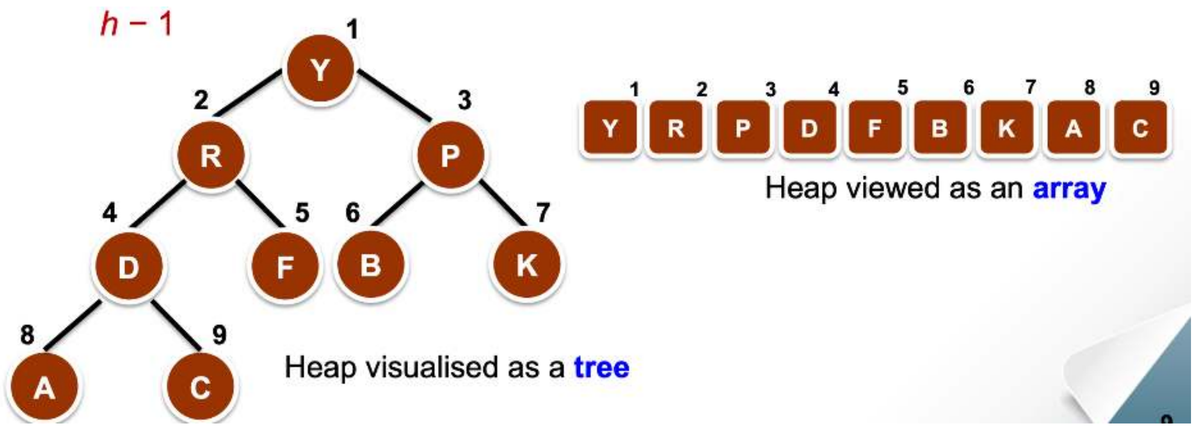

```
int partition(int low, int high)
{
    int i, last_small, pivot;
    int mid = (low+high)/2;
    swap(low, mid);
    pivot = slot[low];
    last_small = low;

        for (i = low+1; i <= high; i++)
                if (slot[i] < pivot)
                        swap(++last_small, i);
        swap(low, last_small);
        return last_small;
}
```

Complexity

Best Case (chosen pivot is perfect middle)	$T(1) = 0$, $T(n) = 2T(n/2) + cn$, where c is a constant $T(n) = 2\,(2T(n/4) + cn/2) + cn$ $= 2^2T(n/4) + 2cn$ $= 2^3T(n/8) + 3cn$ … $= 2^k\,T(n/2^k) + kcn$ $= nT(1) + cn\lg n = cn\lg n$ $\therefore\ T(n) = \Theta(n\lg n)$ • Time complexity: O(nlgn)
Worst Case (chosen pivot is smallest or largest in array)	• Total no. of comparisons: $\sum_{k=2}^{n}(k-1) = \sum_{k=1}^{n-1}k = \frac{n(n-1)}{2}$ • Time complexity: $O(n^2)$
Average Case	• Proof not needed • Time complexity: O(nlgn)

Heap sort



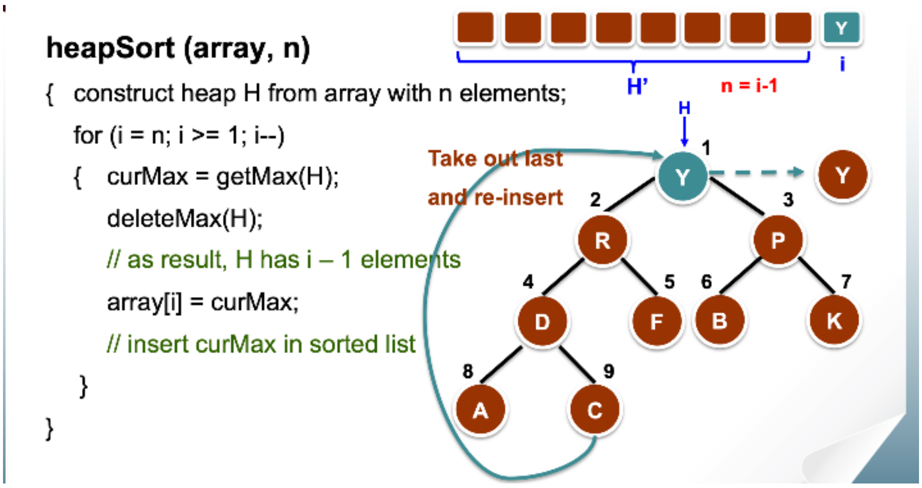
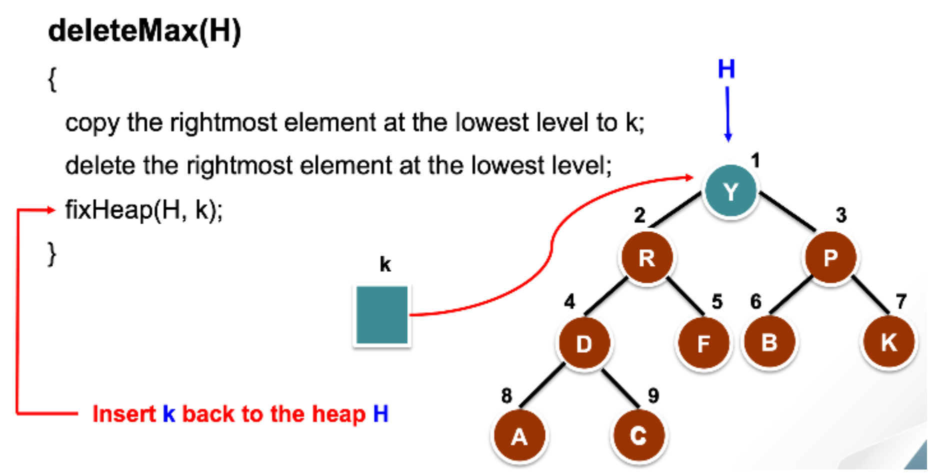
Heap definitions:

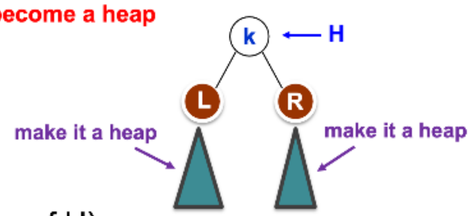
- A binary tree T with height h is a heap structure if and only if it satisfies the following conditions:
  - T is complete at least through depth h − 1
  - All leaves are at depth h or h−1
  - All paths to a leaf of depth h are to the left of all paths to a leaf of depth h−1
- Tree must fill from left to right, all nodes on depth h-1 must be filled before filling depth h
- If parent is i, left subtree = 2i, right subtree = 2i+1

Maximising vs minimising

- Maximising tree: each node has a key value greater than or equal to each of its child nodes
- Minimising tree: each node has a key value smaller than or equal to each of its child nodes

Pseudocode

<table>
<tr><td>HeapSort</td><td>
</td></tr>
<tr><td>DeleteMax</td><td>
</td></tr>
</table>

<table>
<tr><td>fixHeap</td><td>fixHeap(H, k) { // recursive
if (H is a leaf)
insert k in root of H;
else {
compare left child with right child;
largerSubHeap = the larger child of H;
if (k >= key of root(largerSubHeap))
insert k in root of H;
else {
insert root(largerSubHeap) in root of H;
fixHeap(largerSubHeap, k);
}
}
}</td></tr>
<tr><td>ConstructHeap</td><td>constructHeap(array, H)
{
put all elements of array into a heap structure H in arbitrary order;
heapifying(H);
}
Uses the fixheap function mentioned earlier</td></tr>
<tr><td>Heapify</td><td>make binary tree H become a heap
heapifying(H)
{
if (H is not a leaf) {
heapifying(left subtree of H);
heapifying(right subtree of H);
k = root(H);
fixHeap(H, k);
}
}


Post-order traversal of a binary tree
30</td></tr>
</table>

Complexity

<table>
<tr>
<td>fixHeap</td>
<td>

```
fixHeap(H, k)            // recursive
{
  if (H is a leaf)        // Heap has just one node        O(1)
      insert k in root of H;                               O(1)
  else {
1 comparison ⟶ LargerSH = Sub-Heap at larger child of H's root;   O(1)
1 comparison ⟶ if (k >= LargerSH's root key)                      O(1)
          insert k in root of H;                           O(1)
      else {
          insert LargerSH's root key in root of H;         O(1)
          fixHeap(LargerSH, k);
      }
  }
}
```

Each recursive call moves down a level
Total no. of key comparisons ≤ 2 × tree height

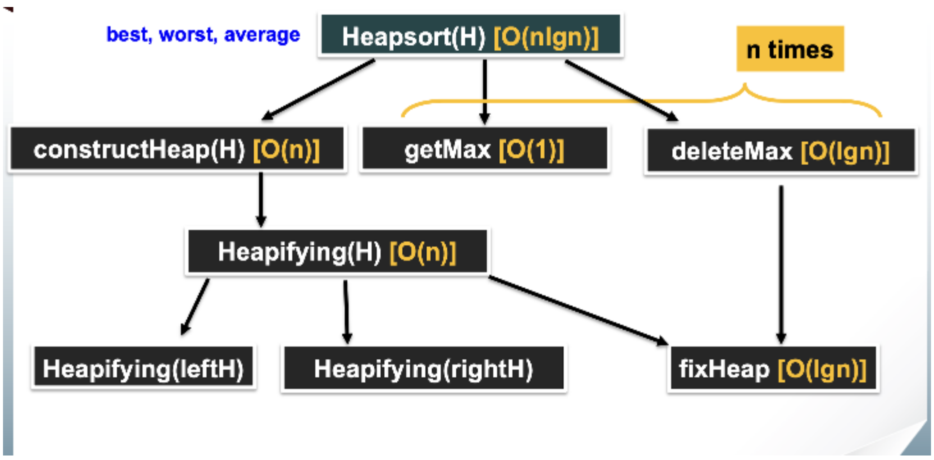
- Height of a heap with n nodes is O(lgn).
- Worst-case time complexity: O(lgn)

</td>
</tr>
<tr>
<td>Heapifying</td>
<td>

```
heapifying(H)                                  W(n)
{
    if (H is not a leaf)                       O(1)
    {
        heapifying(left subtree of H);         W((n-1)/2)
        heapifying(right subtree of H);        W((n-1)/2)
        k = root(H);                           O(1)
        fixHeap(H, k);                         2 lgn
    }
}
```

- Proof not needed
- Worst-case time complexity: O(n)

</td>
</tr>
<tr>
<td>HeapSort</td>
<td>



- Heapsort best/worst case: O(nlgn)
- Heap construction: O(n)

</td>
</tr>
</table>

Summary