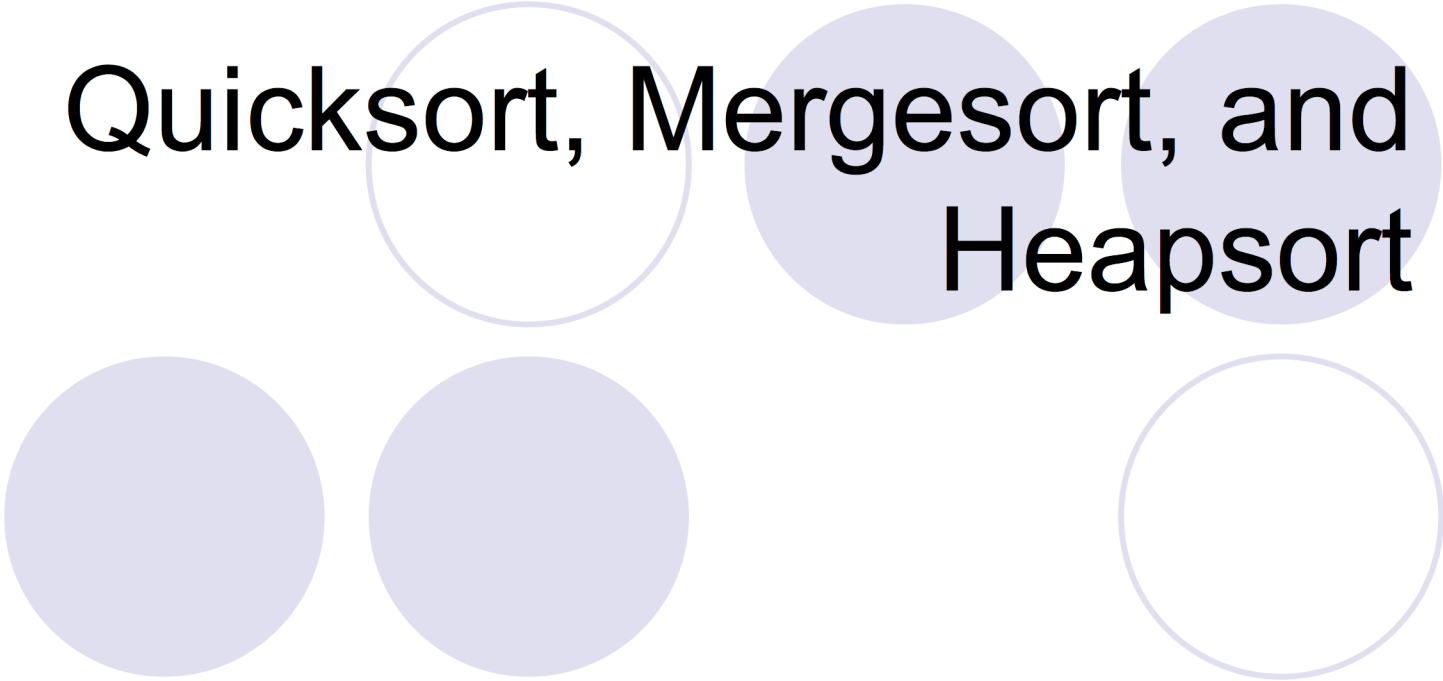
Quicksort, Mergesort, and Heapsort



Quicksort

- Fastest known sorting algorithm in practice
  - Caveats: not stable
  - Vulnerable to certain attacks

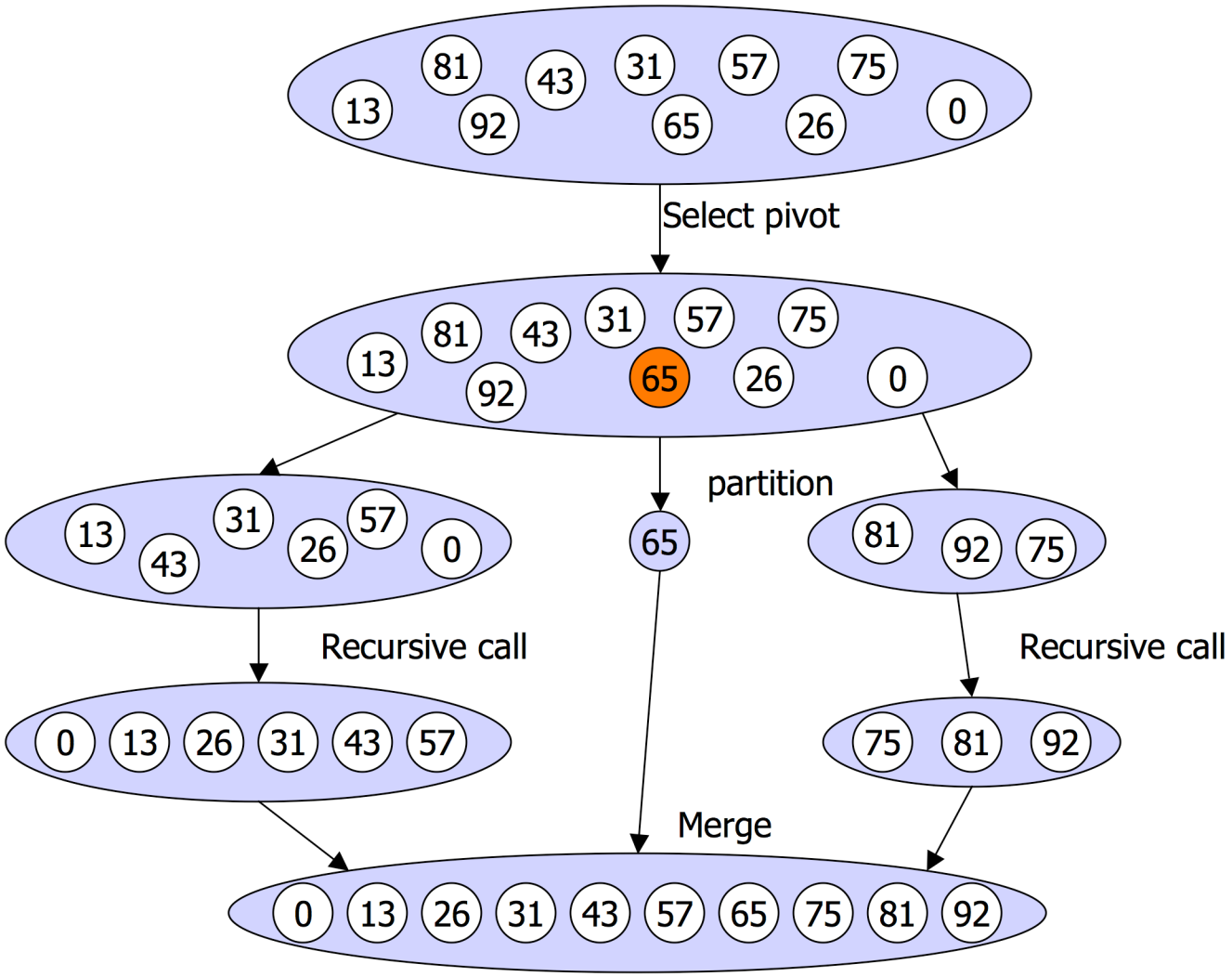
- Average case complexity → $O(N \log N)$

- Worst-case complexity → $O(N^2)$
  - Rarely happens, if coded correctly

Quicksort Outline

- Divide and conquer approach
- Given array `S` to be sorted
  - If size of `S` $\leq$ `1` then done;
  - Pick any element `v` in `S` as the pivot
  - Partition `S-{v}` (remaining elements in `S`) into two groups
    - `S1` = {all elements in `S-{v}` that are smaller than `v`}
    - `S2` = {all elements in `S-{v}` that are larger than `v`}
  - Return {`quicksort(S1)` followed by `v` followed by `quicksort(S2)` }
- Trick lies in handling the partitioning (step 3).
  - Picking a good pivot
  - Efficiently partitioning in-place

Quicksort example



Picking the Pivot

Picking the Pivot

- How would you pick one?

Picking the Pivot

- How would you pick one?
- Strategy 1: Pick the first element in **`S`**
  - Works only if input is random
  - What if input **`S`** is sorted, or even mostly sorted?
    - All the remaining elements would go into either **`S1`** or **`S2`**!
    - Terrible performance!
  - Why worry about sorted input?
    - Remember → Quicksort is recursive, so sub-problems could be sorted
    - Plus mostly sorted input is quite frequent

Picking the Pivot (contd.)

- Strategy 2: Pick the pivot randomly
  - Would usually work well, even for mostly sorted input
  - Unless the random number generator is not quite random!
  - Plus random number generation is an expensive operation

Picking the Pivot (contd.)

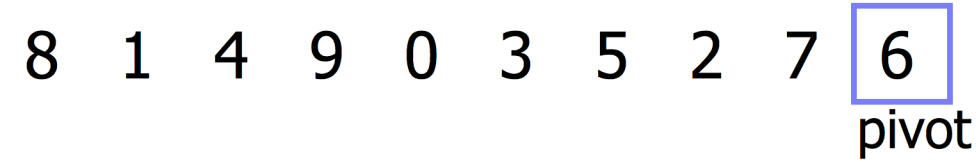
- Strategy 3: Median-of-three Partitioning
  - *Ideally*, the pivot should be the median of input array **s**
    - Median = element in the middle of the sorted sequence
  - Would divide the input into two almost equal partitions
  - Unfortunately, its hard to calculate median quickly, without sorting first!
  - So find the approximate median
    - Pivot = median of the left-most, right-most and center element of the array **s**
    - Solves the problem of sorted input

Picking the Pivot (contd.)

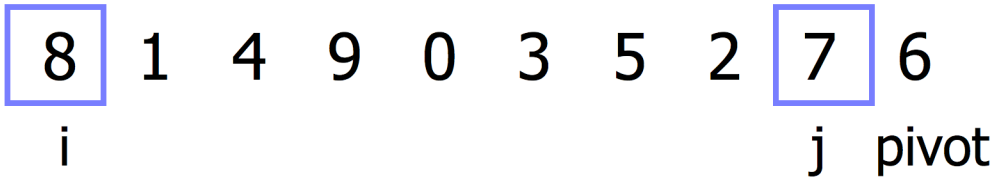
- Example: Median-of-three Partitioning
  - Let input **`S = {6, 1, 4, 9, 0, 3, 5, 2, 7, 8}`**
  - **`left=0`** and **`S[left] = 6`**
  - **`right=9`** and **`S[right] = 8`**
  - **`center = (left+right)/2 = 4`** and **`S[center] = 0`**
  - Pivot
    - = Median of **`S[left]`**, **`S[right]`**, and **`S[center]`**
    - = median of 6, 8, and 0
    - **`= S[left] = 6`**

Partitioning Algorithm

- Original input : `S = {6, 1, 4, 9, 0, 3, 5, 2, 7, 8}`
- Get the pivot out of the way by swapping it with the last element



- Have two 'iterators' – `i` and `j`
  - `i` starts at first element and moves forward
  - `j` starts at last element and moves backwards

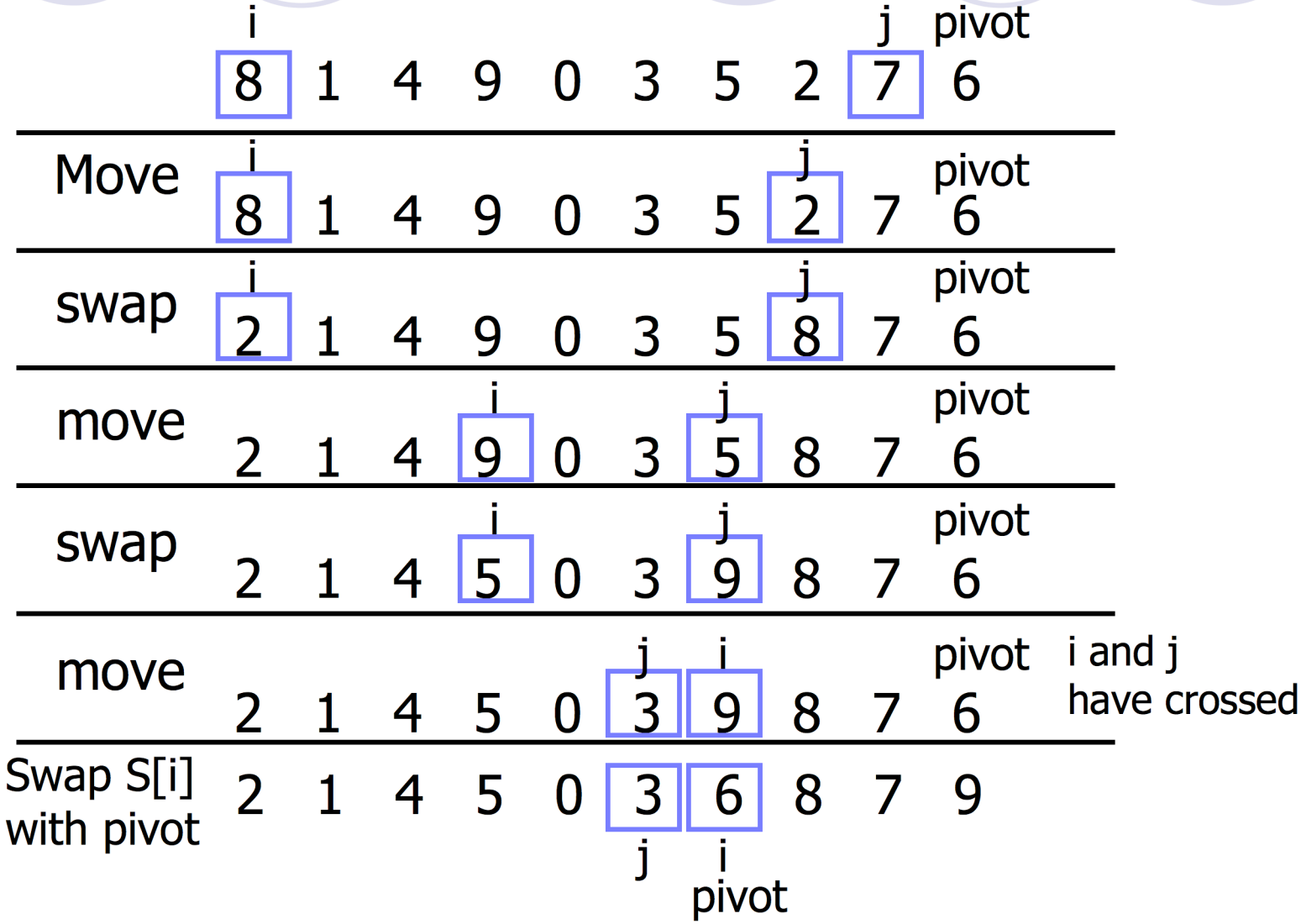


Partitioning Algorithm (contd.)

- **`While (i < j)`**
  - Move **`i`** to the right till we find a number greater than **`pivot`**
  - Move **`j`** to the left till we find a number smaller than **`pivot`**
  - **`If (i < j) swap(S[i], S[j])`**
  - (The effect is to push larger elements to the right and smaller elements to the left)

Swap the **`pivot`** with **`S[i]`**

Partitioning Algorithm Illustrated



Dealing with small arrays

- For small arrays **(N ≤ 20)**,
  - Insertion sort is faster than quicksort
- Quicksort is recursive
  - So it can spend a lot of time sorting small arrays
- Hybrid algorithm:
  - Switch to using insertion sort when problem size is small (say for **N < 20**)

Quicksort Driver Routine

```
1   /**
2    * Quicksort algorithm (driver).
3    */
4   template <typename Comparable>
5   void quicksort( vector<Comparable> & a )
6   {
7       quicksort( a, 0, a.size( ) - 1 );
8   }
```

Quicksort Pivot Selection Routine

```
1   /**
2    * Return median of left, center, and right.
3    * Order these and hide the pivot.
4    */
5   template <typename Comparable>
6   const Comparable & median3( vector<Comparable> & a, int left, int right )
7   {
8       int center = ( left + right ) / 2;
9       if( a[ center ] < a[ left ] )
10          swap( a[ left ], a[ center ] );
11      if( a[ right ] < a[ left ] )
12          swap( a[ left ], a[ right ] );
13      if( a[ right ] < a[ center ] )
14          swap( a[ center ], a[ right ] );
15
16          // Place pivot at position right - 1
17      swap( a[ center ], a[ right - 1 ] );
18      return a[ right - 1 ];
19  }
```

Swap a[left], a[center] and a[right] in-place

Pivot is in a[center] now

Swap the pivot a[center] with a[right-1]

Quicksort routine

```
1   /**
2    * Internal quicksort method that makes recursive calls.
3    * Uses median-of-three partitioning and a cutoff of 10.
4    * a is an array of Comparable items.
5    * left is the left-most index of the subarray.
6    * right is the right-most index of the subarray.
7    */
8   template <typename Comparable>
9   void quicksort( vector<Comparable> & a, int left, int right )
10  {
11      if( left + 10 <= right )
12      {
13          Comparable pivot = median3( a, left, right );
14
15              // Begin partitioning
16          int i = left, j = right - 1;
17          for( ; ; )
18          {
19              while( a[ ++i ] < pivot ) { }
20              while( pivot < a[ --j ] ) { }
21              if( i < j )
22                  swap( a[ i ], a[ j ] );
23              else
24                  break;
25          }
26
27          swap( a[ i ], a[ right - 1 ] );  // Restore pivot
28
29          quicksort( a, left, i - 1 );     // Sort small elements
30          quicksort( a, i + 1, right );    // Sort large elements
31      }
32      else  // Do an insertion sort on the subarray
33          insertionSort( a, left, right );
34  }
```

Has a side effect

move

swap

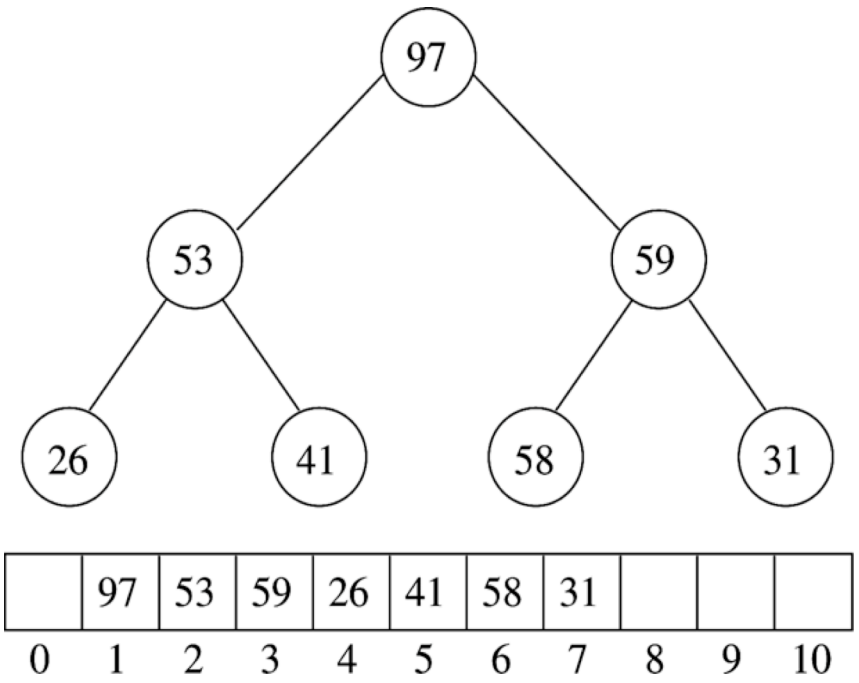
Exercise: Runtime analysis

- Worst-case
- Average case
- Best case

Heapsort

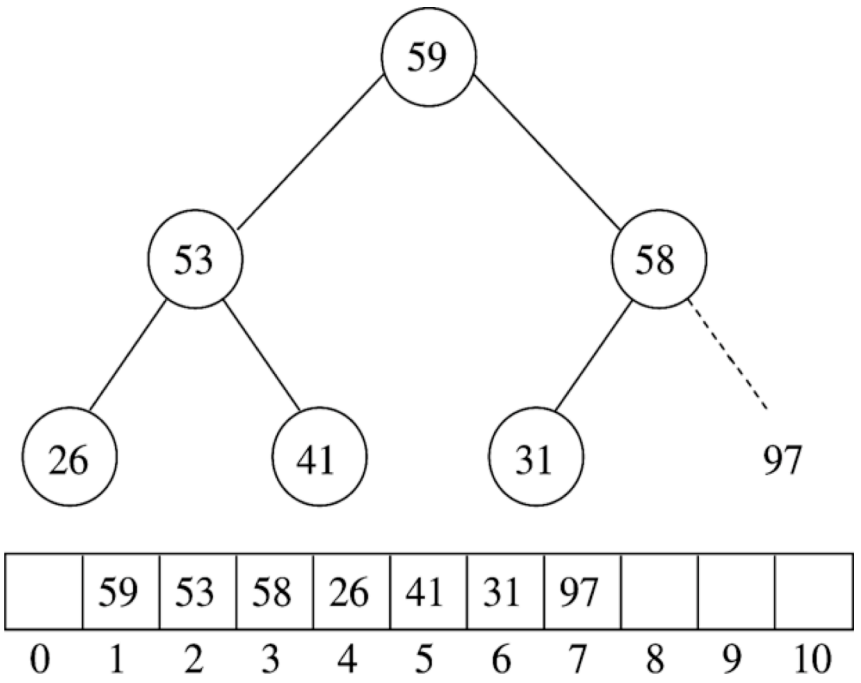
- Build a binary heap of N elements
  - ***O(N)*** time
- Then perform ***N*** **`deleteMax`** operations
  - ***log(N)*** time per **`deleteMax`**
- Total complexity ***O(N log N)***

Example



	97	53	59	26	41	58	31			
0	1	2	3	4	5	6	7	8	9	10

After BuildHeap



	59	53	58	26	41	31	97			
0	1	2	3	4	5	6	7	8	9	10

After first deleteMax

Heapsort Implementation

```
1   /**
2    * Standard heapsort.
3    */
4   template <typename Comparable>
5   void heapsort( vector<Comparable> & a )
6   {
7       for( int i = a.size( ) / 2; i >= 0; i-- )
8           percDown( a, i, a.size( ) );
9       for( int j = a.size( ) - 1; j > 0; j-- )
10      {
11          swap( a[ 0 ], a[ j ] );
12          percDown( a, 0, j );
13      }
14  }
15
16  /**
17   * Internal method for heapsort.
18   * i is the index of an item in the heap.
19   * Returns the index of the left child.
20   */
21  inline int leftChild( int i )
22  {
23      return 2 * i + 1;
24  }
25
```

```
26  /**
27   * Internal method for heapsort that is used in deleteMax and buildHeap.
28   * i is the position from which to percolate down.
29   * n is the logical size of the binary heap.
30   */
31  template <typename Comparable>
32  void percDown( vector<Comparable> & a, int i, int n )
33  {
34      int child;
35      Comparable tmp;
36
37      for( tmp = a[ i ]; leftChild( i ) < n; i = child )
38      {
39          child = leftChild( i );
40          if( child != n - 1 && a[ child ] < a[ child + 1 ] )
41              child++;
42          if( tmp < a[ child ] )
43              a[ i ] = a[ child ];
44          else
45              break;
46      }
47      a[ i ] = tmp;
48  }
```

Mergesort

- Divide the ***N*** values to be sorted into two halves
- Recursively sort each half *using Mergesort*
  - Base case ***N=1*** → no sorting required
- Merge the two halves
  - ***O(N)*** operation
- Complexity??
  - We'll see

Mergesort Implementation

```
1   /**
2    * Mergesort algorithm (driver).
3    */
4   template <typename Comparable>
5   void mergeSort( vector<Comparable> & a )
6   {
7       vector<Comparable> tmpArray( a.size( ) );
8
9       mergeSort( a, tmpArray, 0, a.size( ) - 1 );
10  }
11
12  /**
13   * Internal method that makes recursive calls.
14   * a is an array of Comparable items.
15   * tmpArray is an array to place the merged result.
16   * left is the left-most index of the subarray.
17   * right is the right-most index of the subarray.
18   */
19  template <typename Comparable>
20  void mergeSort( vector<Comparable> & a,
21                  vector<Comparable> & tmpArray, int left, int right )
22  {
23      if( left < right )
24      {
25          int center = ( left + right ) / 2;
26          mergeSort( a, tmpArray, left, center );
27          mergeSort( a, tmpArray, center + 1, right );
28          merge( a, tmpArray, left, center + 1, right );
29      }
30  }
```

Complexity analysis

- $T(N) = 2T(N/2) + N$
  - Recurrence relation
- $T(N) = 4T(N/4) + 2N$
- $T(N) = 8T(N/8) + 3N$
- …
- $T(N) = 2^kT(N/2^k) + k*N$

- For $k = \log N$
  - $T(N) = N\,T(1) + N \log N$