Internal Sorting Algorithms

A sorting algorithm can be classified as being **internal** if the records that it is sorting are in main memory, or **external** if some of the records that it is sorting are in secondary storage.

In this course, we shall only discuss internal sorting algorithms.
To simplify discussion, sorting of an integer array is used in our examples.

Efficiency of a sorting method is usually measured by the number of comparisons and data movements required.

Insertion sort

Successively insert a new element into a sorted sublist.

Example:

input array: [25 57 48 37 92 60]

sorted sublist: [25] ; initial condition

sorted sublist: [25 57] ; insert 57

sorted sublist: [25 48 57] ; insert 48

sorted sublist: [25 37 48 57] ; insert 37

sorted sublist: [25 37 48 57 92] ; insert 92

sorted sublist: [25 37 48 57 60 92] ; insert 60

```
void insertionSort(int x[], int N)
{
   for (int i = 1; i < N; i++)
   {
      int t = x[i];
      int j;
      for (j = i-1; j >= 0 && x[j] > t; j--)
         x[j+1] = x[j];

      x[j+1] = t;
   }
}
```

Complexity of insertion sort

- Let c be the number of comparisons required to insert an element into a sorted sublist of size k.

Best case:	$c = 1$
Average case:	$c = k/2$
Worst case:	$c = k$

- The total number of comparisons = $\sum_{k=1}^{N-1} c$

- Overall time complexity

Best case	$c = 1$	$O(N)$
Average case	$c = k/2$	$O(N^2)$
Worst case	$c = k$	$O(N^2)$

- Number of data movement = number of comparisons

- Because of the simplicity of insertion sort, it is the fastest sorting method when the number of elements N is small, e.g. $N < 10$.

Generic sorting function for any data type

```
template<class Type>
void insertionSort(Type *x, unsigned N,
                   int (*compare)(const Type&, const Type&))
{
   for (int i = 1; i < N; i++)
   {
      Type t = x[i];
      int j;
      for (j = i-1; j >= 0 && (*compare)(x[j], t) > 0; j--)
         x[j+1] = x[j];

      x[j+1] = t;
   }
}
```

Quicksort

Consider an array $x[0..N-1]$

1. Choose a pivot element a from a specific position, say $a=x[0]$
2. partition $x[]$ using a, *i.e.* a is placed into position j (*i.e.* $x[j] = a$) such that $x[i] \leq a$ for $i = 1, 2, \ldots, (j-1)$, and $x[k] > a$ where $k = j+1, \ldots, N\text{-}1$
3. The two subarrays $x[0..(j-1)]$ and $x[(j+1)..N-1]$ are sorted recursively using the same method.

Partitioning operation:

Step 1: scan the array from left to right to look for an element x[i] > x[0]

Step 2: scan the array from right to left to look for an element x[j] ≤ x[0]

Step 3: if ($i < j$) swap x[i] with x[j] and go to step 1;
otherwise swap x[0] with x[j] (partitioning finished).

Example: Partitioning process

```
   |→                      ←|
50  25  57  48  37  92  33  64      ;swap x[i] with x[j]
        ↑i                  ↑j

50  25  33  48  37  92  57  64      ;swap x[0] with x[j]
                ↑j  ↑i

37  25  33  48  50  92  57  64      ;partitioning finished
[←  ≤  50   →]      [←> 50 →]       ;sort the left and right sublists recursively
  left sublist      right sublist
```

```
void swap(int x[], int i, int j)
{
   int t = x[i];
   x[i] = x[j];
   x[j] = t;
}


int partition(int x[], int lb, int ub)
{
   int i = lb;
   int j = ub;
   bool done = false;

   while (!done)
   {
      while (i < j && x[i] <= x[lb])
         i++;

      while (x[j] > x[lb])  //j will NOT go out of bound
         j--;

      if (i < j)
         swap(x, i, j);
      else
         done = true;
   }
   swap(x, lb, j); // swap x[lb] and x[j]

   retrun j;
}


void simpleQuicksort(int x[], int start, int end)
{
   if (start < end)
   {
      int j = partition(x, start, end);
      simpleQuicksort(x, start, j-1);
      simpleQuicksort(x, j+1, end);
   }
}
```

Complexity of quicksort

Let $T(N)$ be the time to sort an array of size N using quicksort, and $T(1) = b$.
The time to partition the array = cN.
b and c are some constants.

In the best case, each time an array is partitioned into 2 subarrays of roughly equal size.

$$T(N) = 2T(N/2) + cN$$

We can expand the recurrent equation:

$$\begin{aligned} T(N) &= 2(2T(N/4) + cN/2) + cN \\ &= 4T(N/4) + cN + cN \\ &= \ldots \\ &= NT(1) + cN + \ldots + cN \\ &= bN + (\log_2 N) \times cN \\ &= O(N \log_2 N) \end{aligned}$$

In the worst case, each time an array is partitioned into an empty subarray and a subarray of size N–1.

$$\begin{aligned} T(N) &= T(N-1) + cN \\ &= T(N-2) + c(N-1) + cN \\ &= \ldots \\ &= T(1) + 2c + \ldots + cN \\ &= b + c \sum_{i=2}^{N} i \\ &= O(N^2) \end{aligned}$$

It can also be shown that the average cast complexity of quicksort is approximately equal to 1.38 $N \log_2 N$.

Remark: If the size of the array is large, quicksort is the fastest sorting method known today.

An improved version of quicksort

1. Choose the median of the first, last, and middle elements as the pivot in the partitioning process.
2. If the length of the sublist is less than some threshold, e.g. 10, use insertion sort to sort the sublist instead.

```
#define Threshold 10

void quicksort(int x[], int start, int end)
{
   if (start < end - Threshold)
   {
      int mid = (start + end) / 2;

      if (x[start] > x[mid])
      {
         if (x[mid] > x[end])
            swap(x, mid, start);  // order: s > m > e
         else if (x[start] > x[end])
            swap(x, start, end);  // order: s > e > m
      }
      else
      {
         if (x[end] > x[mid])
            swap(x, start, mid);  // order: e > m > s
         else if (x[end] > x[start])
            swap(x, start, end);  // order: m > e > s
      }

      //assert: x[start] is the median of x[start], x[mid],
      //        and x[end]

      int j = partition(x, start, end);
      quicksort(x, start, j-1);
      quicksort(x, j+1, end);
   }
   else
      // do insertSort for small input size
      insertionSort(x, start, end); //slight modification
                                    //required
}
```

The `qsort()` function in the C library `<cstdlib>`

```
//function signature
void qsort(void *base, unsigned num, unsigned objSize,
           int (*compare)(const void *, const void *));

// void * is a pure address, with no data type information
// associated to it.


//Example: use qsort()to sort an array of fraction

int compareFraction(const void *a, const void *b)
{
   fraction *f1 = (fraction *)a;  //type cast the pointer
   fraction *f2 = (fraction *)b;  //before using it to refer
                                  //to an object
   if (*f1 == *f2)
      return 0;

   if (*f1 < *f2)
      return -1;

   return 1;
}


int main()
{
   int len = 100;
   fraction *list = new fraction[len];

   // codes to assign values to list[];

   qsort(list, len, sizeof(fraction), compareFraction);

}
```

Use the `qsort` function to sort a list of names (`cstring, char []`)

```
// the void pointer arguments point to cstring (char*)
// i.e. (char**), which is pointer-to-(char*)
int compareString(const void *a, const void *b)
{
   char **c1 = (char **)a;
   char **c2 = (char **)b;

   // dereferencing once becomes cstring (char *)
   return strcmp(*c1, *c2);  //compare cstring
}

int main()
{
   char *name[] = {"Wong Chi Ming",
                   "Chan Tai Man",
                   "Ho Pui Shan",
                   "Au Pui Ki",
                   "Cheung Ka Man"};


   qsort(name, 5, sizeof(char *), compareString);
}
```

Use insertion sort to illustrate the internal details of the sort function in the C library `<cstdlib>`

Function `memcpy` (memory copy) in `<cstring>`

```
void * memcpy(void *destination, const void *source, unsigned n);
//copy n bytes from source to destination
```

```
void insertionSort(void *base, unsigned n, unsigned objSize,
                   int (*compare)(const void *, const void *))
{
   char *t = new char[objSize];
   char *b = (char *)base;

   for (int i = 0; i < n; i++)
   {
      memcpy(t, b+i*objSize, objSize); //t = base[i]

      int j;
      for (j = i-1; j > = 0 && (*compare)(b+j*objSize, t) > 0;
           j--)
         memcpy(b+(j+1)*objSize, b+j*objSize, objSize);
         //base[j+1] = base[j]

      memcpy(b+(j+1)*objSize, t, objSize);  //base[j+1] = t

   }
   delete [] t;
}
```

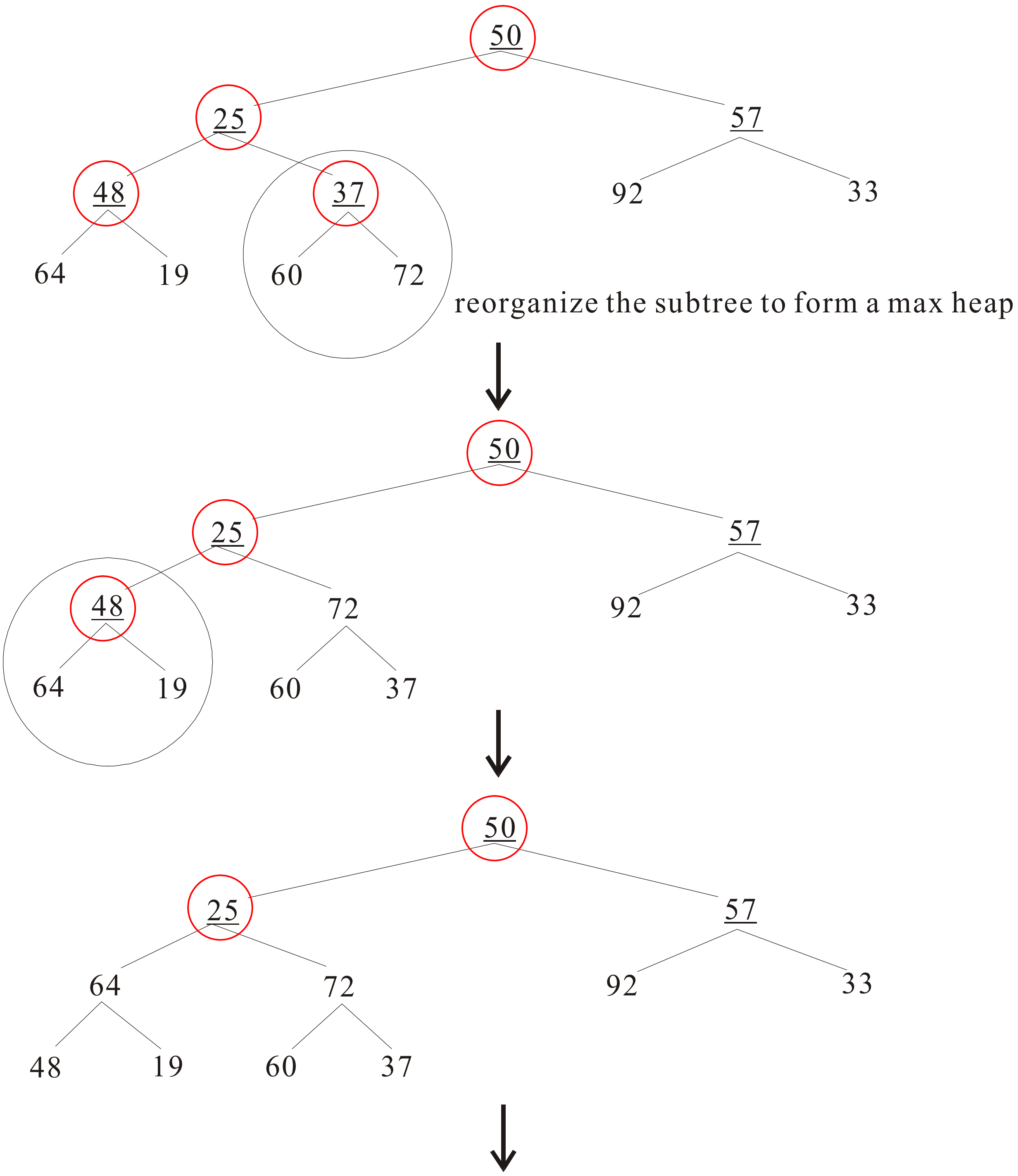
Heapsort

1. Organize the input array of numbers as a max heap.
2. Iteratively remove the root of the heap (the largest value in the subgroup of numbers to be sorted) and re-adjust the remaining numbers to form a max heap.

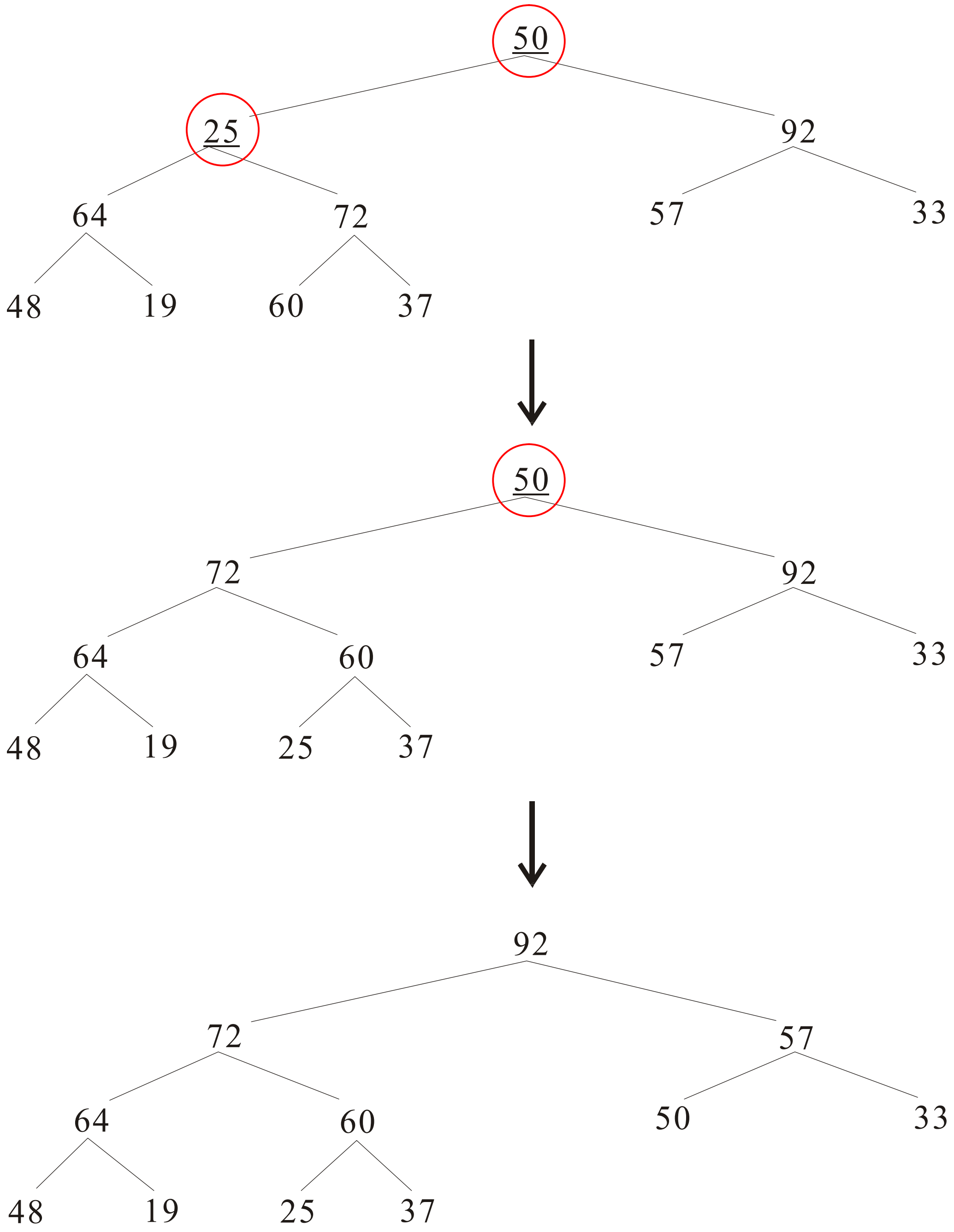
Steps to organize the input array to form a max heap

input array: [50 25 57 48 37 92 33 64 19 60 72]

processing order ← | [← leaf nodes →]



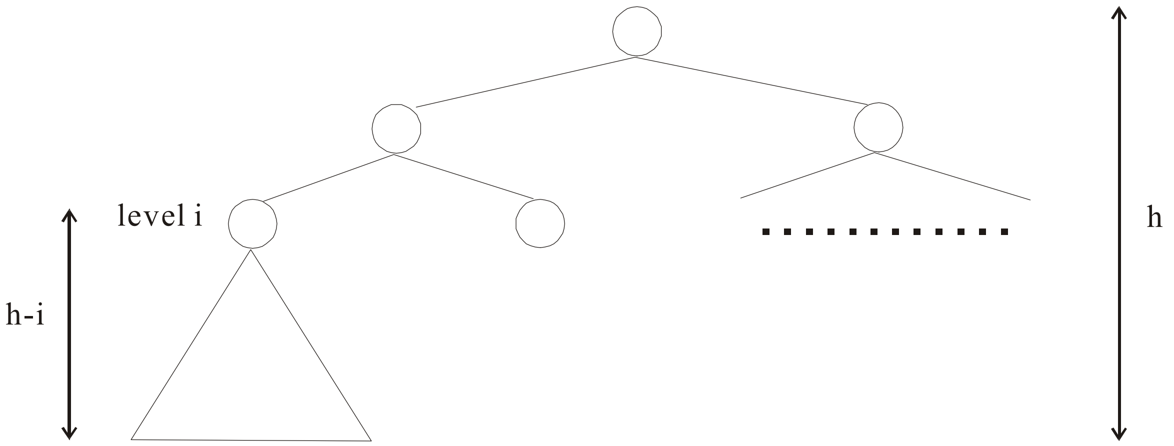
reorganize the subtree to form a max heap


50
25
92
64
72
57
33
48
19
60
37
50
72
92
64
60
57
33
48
19
25
37
92
72
57
64
60
50
33
48
19
25
37

```
void adjust(int x[], int i, int n)
// Adjust the binary tree with root i to satisfy the heap
// property. No node in the tree has index >= n.
{
   int done = 0;
   int value = x[i];
   int j = 2*i + 1;  // j is a son of i

   while (j < n && !done)
   {
      if (j < n-1)  // j has a right brother
         if (x[j] < x[j+1])
            j = j+1;  // j is the larger son of i

      if (value >= x[j])
         done = 1;
      else  // move x[j] to its parent's position
      {
         x[(j-1)/2] = x[j];
         j = 2*j + 1;
      }
   }
   x[(j-1)/2] = value;
}


void heapsort(int x[], int n)
{
   // Phase 1: organize the array to form a max heap
   for (int i = ((n-1)-1)/2; i >= 0; i--)
      adjust(x, i, n);

   // Phase 2: sort the array into ascending order
   for (int i = n-1; i > 0; i--)
   {
      swap(x, 0, i);
      adjust(x, 0, i);  // adjust the remaining tree of
                        // size i to form a max heap
   }
}
```

Complexity of heapsort

Phase 1: Time to organize the array to form a max heap:



Suppose root node is on level 1
h = height of the tree
n = number of nodes in the tree, $2^{h-1} \le n \le 2^h$
number of nodes on level $i = 2^{i-1}$

$$\text{Total number of swaps} = \sum_{i=1}^{h} (h-i)2^{i-1}$$

$$= \sum_{j=0}^{h-1} j2^{h-j-1} \qquad \text{(substitute j} = h\text{–}i\text{)}$$

$$= 2^{h-1} \sum_{j=0}^{h-1} \frac{j}{2^j}$$

$$< n \sum_{j=0}^{h-1} \frac{j}{2^j}$$

$$< 2n \qquad \text{(because } \sum_{j=0}^{\infty} \frac{j}{2^j} = 2 \text{)}$$

So, the complexity for forming a max heap is $O(n)$

Phase 2: Time to sort the array = $O(n \log n)$

Total time of heapsort = time of phase 1 + time of phase 2
= $O(n) + O(n \log n)$
= $O(n \log n)$

Merge sort

- Initially the input file is divided into *N* subfiles of size 1.
- Adjacent pairs of files are merged to form larger subfiles.
- The merging process is repeated until there is only one file remaining.

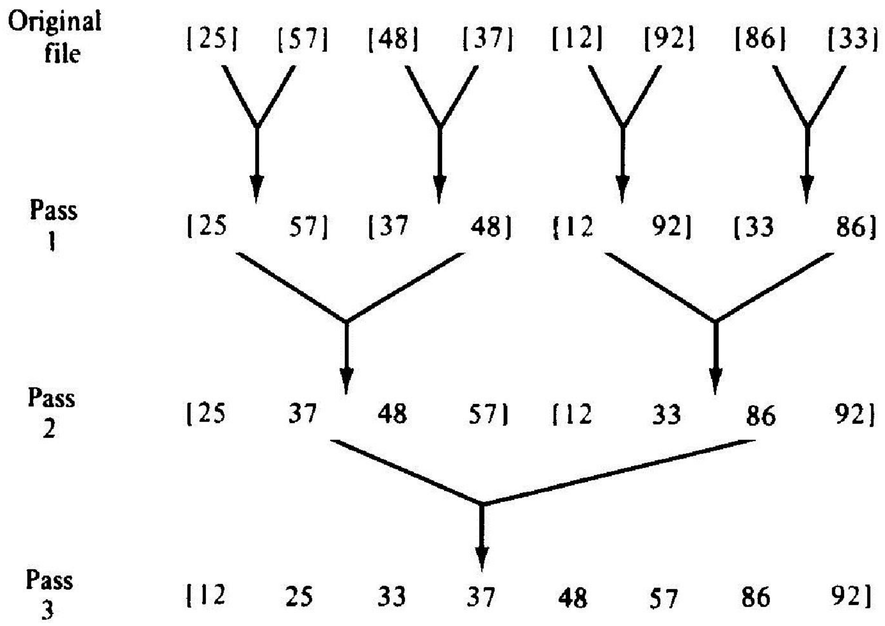


Figure 6.5.1 Successive passes of the merge sort.

pseudo code of the mergesort algorithm

```
//x[] is the input array, aux[] is the temporary storage

size = 1;  //initial size of a subfile
while (size < n) //still have 2 or more subfiles
{
   for (i = 0; i < no. of subfiles - 1; i += 2)
      merge subfiles i and i+1 from x[] to aux[];

   copy any remaining single subfile from x[] to aux[];

   size *= 2;

   copy aux[] to x[] for preparation of the next merge-pass;

}
```

```
void mergesort(int x[], int n)
{
   int *aux, i, j, k, L1, L2, u1, u2, size;

   aux = new int[n];

   size = 1; //size of subfiles
   while (size < n)
   {
      L1 = 0; k = 0;
      while (L1 + size < n) // 2 or more files to merge
      {
         L2 = L1 + size;
         u1 = L2 - 1;
         u2 = (L2+size-1 < n) ? L2+size-1 : n-1;

         // merge subfiles x[L1..u1] and x[L2..u2]
         for (i = L1, j = L2; i <= u1 && j <= U2; k++)
            if (x[i] <= x[j])
               aux[k] = x[i++];
            else
               aux[k] = x[j++];

         while (i <= u1)
            aux[k++] = x[i++];
         while (j <= u2)
            aux[k++] = x[j++];

         // advance L1 to the start of the next pair of
         // files
         L1 = u2+1;
      }

      // copy any remaining single file
      for (i = L1; i < n; i++)
        aux[k++] = x[i];

      // copy aux[] back to x[] and adjust size
      for (i = 0; i < n; i++)
         x[i] = aux[i];

      size *= 2;
   }
   delete[] aux;
}
```

Improvement to the mergesort algorithm:

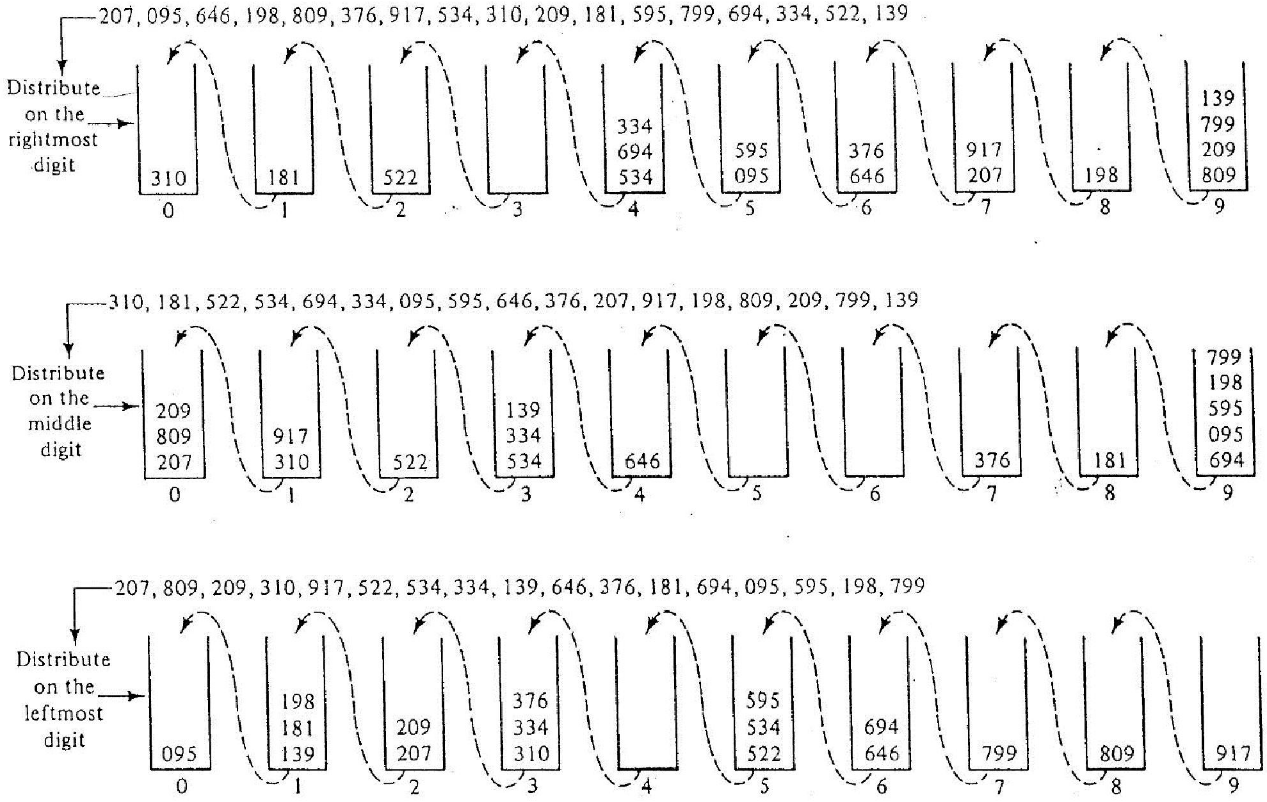
- Instead of merging each set of files from `x[]` to `aux[]` and then copy `aux[]` back to `x[]`, alternate merge passes can be performed from `x[]` to `aux[]` and from `aux[]` to `x[]`.

Complexity of mergesort

- mergesort requires $O(N)$ additional space for the auxiliary array
- The time to do one merge pass is $O(N)$.
- In one merge pass, the size of the sorted subfiles is doubled. Hence, $\log_2 N$ merge passes are required.
- The overall time complexity is $O(N \log_2 N)$.

Radix sort

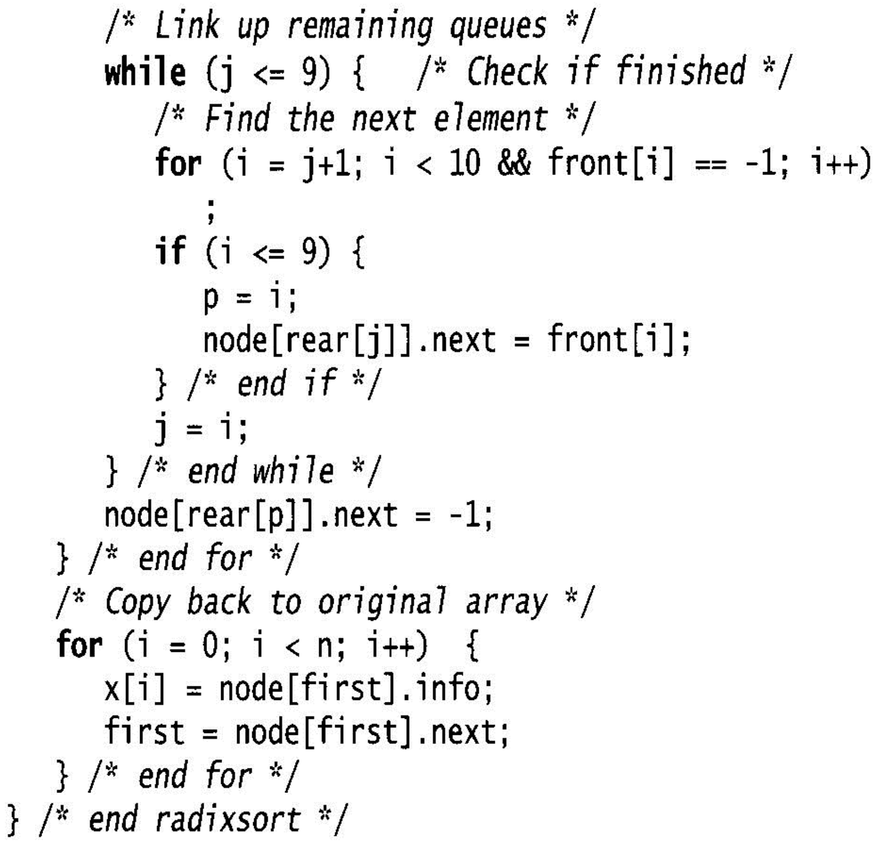
- The key value is interpreted as being consisted of a string of digits.
- Digits are processed from right to left (least significant digit first).
- The numbers to be sorted are distributed to buckets based on the least significant digit.
- Numbers in the 10 buckets are concentrated to form a new list. They will be redistributed based on the next digit.
- The process is repeated until the most significant digit has been processed.



Sorted table: 095, 139, 181, 198, 207, 209, 310, 334, 376, 522, 534, 595, 646, 694, 799, 809, 917

```
for (k = least significant digit; k <= most significant digit; k++) {
   for (i = 0; i < n; i++) {
      y = x[i];
      j = kth digit of y;
      place y at rear of queue[j];
   }  /* end for */

   for (qu = 0; qu < 10; qu++)
      place elements of queue[qu] in next sequential position of x;
}  /* end for */
```

```
void radixsort(int x[], int n)
{
   int front[10], rear[10];
   struct {
      int info;
      int next;
   } node[NUMELTS];
   int exp, first, i, j, k, p, q, y;

   /* Initialize linked list */
   for (i = 0; i < n-1; i++) {
      node[i].info = x[i];
      node[i].next = i+1;
   } /* end for */
   node[n-1].info = x[n-1];
   node[n-1].next = -1;
   first = 0;   /* first is the head of the linked list */
   for (k = 1; k < 5; k++) {
      /* Assume we have four-digit numbers */
      for (i = 0; i < 10; i++) {
         /* Initialize queues */
         rear[i] = -1;
         front[i] = -1;
      } /* end for */
      /* Process each element on the list */
      while (first != -1) {
         p = first;
         first = node[first].next;
         y = node[p].info;
         /* Extract the kth digit */
         exp = power(10,k-1);        /* raise 10 to (k-1)th power */
         j = (y/exp)%10;
         /* Insert y into queue[j] */
         q = rear[j];
         if (q == -1)
            front[j] = p;
         else
            node[q].next = p;
         rear[j] = p;
      } /* end while */
      /* At this point each record is in its proper queue based on digit k. We n
      /* form a single list from all the queue elements. Find the first element.
      for (j = 0; j < 10 && front[j] == -1; j++)
         ;
      first = front[j];
```



```
      /* Link up remaining queues */
      while (j <= 9) {   /* Check if finished */
         /* Find the next element */
         for (i = j+1; i < 10 && front[i] == -1; i++)
            ;
         if (i <= 9) {
            p = i;
            node[rear[j]].next = front[i];
         } /* end if */
         j = i;
      } /* end while */
      node[rear[p]].next = -1;
   } /* end for */
   /* Copy back to original array */
   for (i = 0; i < n; i++)  {
      x[i] = node[first].info;
      first = node[first].next;
   } /* end for */
} /* end radixsort */
```

Complexity of radix sort:

Assume a number has *k* digits and the base of each digit is *b*.

Time to sort *N* numbers = $O(k(N+b))$

If $N >> b$, then the complexity of radix sort is $O(kN)$.

Stable property

- Let *a* and *b* be 2 records with equal key value, and *a* appears before *b* in the input list.
- A sorting method is said to be **stable** if the relative order of *a* and *b* is preserved in the output, i.e. *a* appears before *b* in the sorted list.
- Bubble sort, insertion sort, mergesort and radix sort are stable
- quicksort and heapsort are not stable.