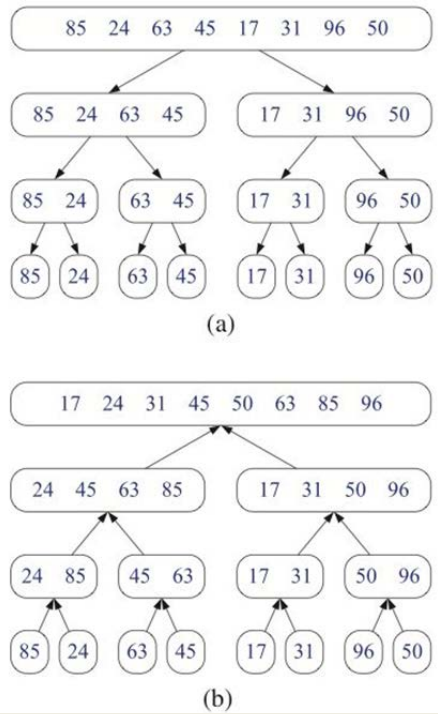
CH11. SORTING

CSED233 Data Structure

Prof. Hwanjo Yu
POSTECH

Merge-Sort

- Sorting: order n objects according to a comparator
- Divide-and-Conquer
  - *Divide the input data into two or more disjoint subsets until the input size is small enough to be solved using a straightforward method*
  - *Use the solutions on the subset or the subproblems to build a solution to its "parent" problem*
- Merge-Sort

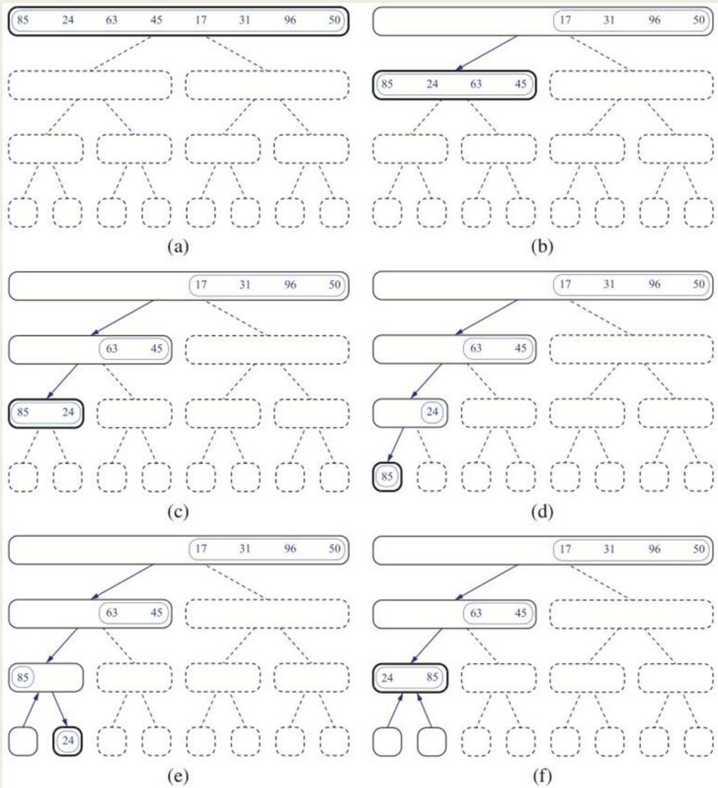


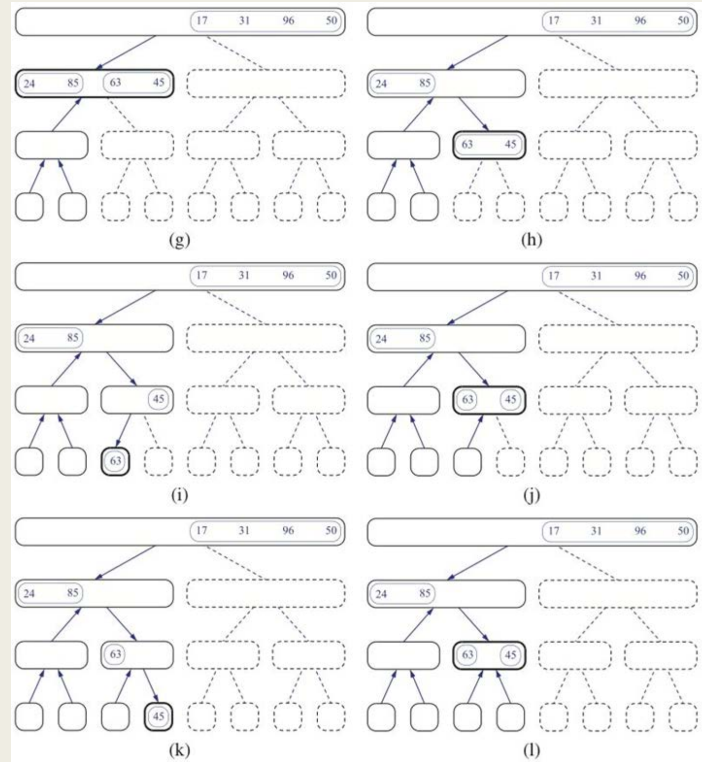
(a)
(b)

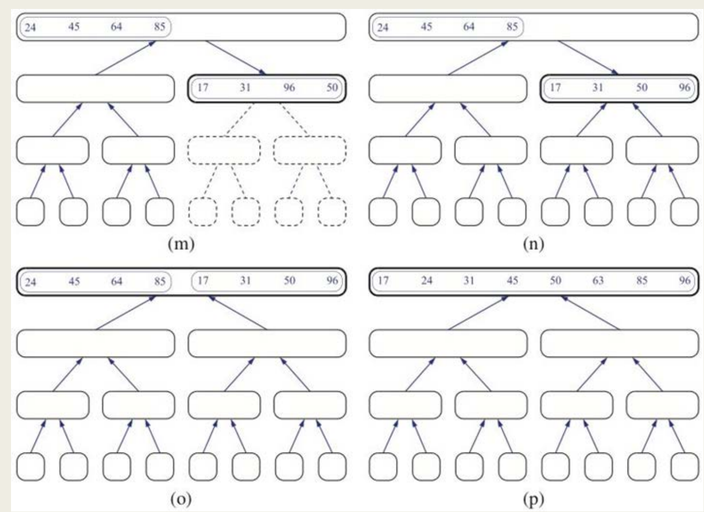
```
template <typename E, typename C>              // merge-sort S
void mergeSort(list<E>& S, const C& less) {
  typedef typename list<E>::iterator Itor;     // sequence of elements
  int n = S.size();
  if (n <= 1) return;                          // already sorted
  list<E> S1, S2;
  Itor p = S.begin();
  for (int i = 0; i < n/2; i++) S1.push_back(*p++); // copy first half to S1
  for (int i = n/2; i < n; i++) S2.push_back(*p++); // copy second half to S2
  S.clear();                                   // clear S's contents
  mergeSort(S1, less);                         // recur on first half
  mergeSort(S2, less);                         // recur on second half
  merge(S1, S2, S, less);                      // merge S1 and S2 into S
}

template <typename E, typename C>              // merge utility
void merge(list<E>& S1, list<E>& S2, list<E>& S, const C& less) {
  typedef typename list<E>::iterator Itor;     // sequence of elements
  Itor p1 = S1.begin();
  Itor p2 = S2.begin();
  while(p1 != S1.end() && p2 != S2.end()) {    // until either is empty
    if(less(*p1, *p2))                         // append smaller to S
      S.push_back(*p1++);
    else
      S.push_back(*p2++);
  }
  while(p1 != S1.end())                        // copy rest of S1 to S
    S.push_back(*p1++);
  while(p2 != S2.end())                        // copy rest of S2 to S
    S.push_back(*p2++);
}
```

Merge-Sort







Merge-Sort

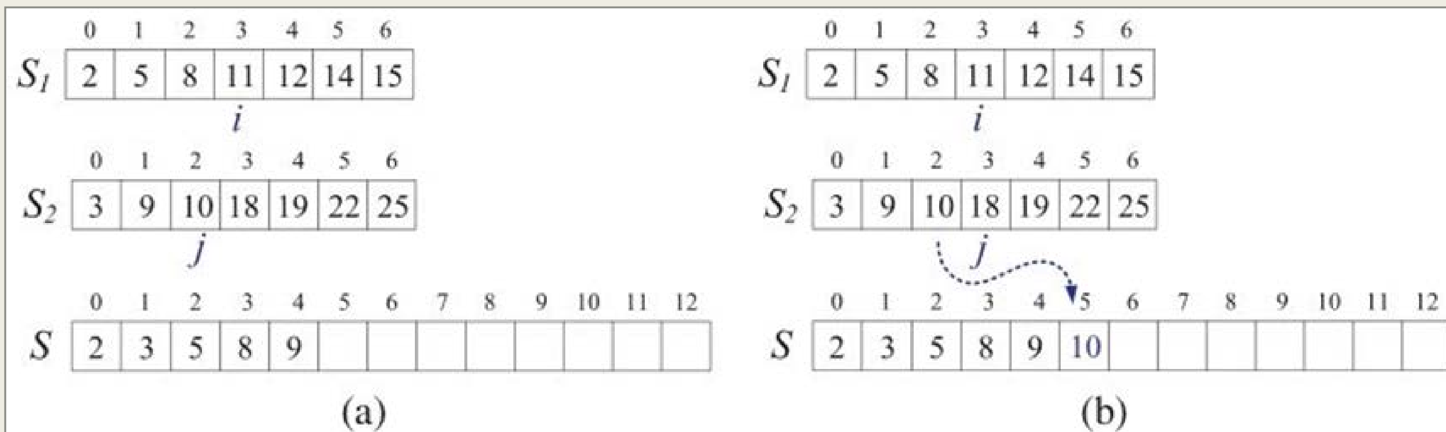
Algorithm merge(S_1, S_2, S):

Input: Sorted sequences S_1 and S_2 and an empty sequence S, all of which are implemented as arrays

Output: Sorted sequence S containing the elements from S_1 and S_2

$i \leftarrow j \leftarrow 0$
while $i < S_1$.size() **and** $j < S_2$.size() **do**
  if $S_1[i] \leq S_2[j]$ **then**
    S.insertBack($S_1[i]$) {copy ith element of S_1 to end of S}
    $i \leftarrow i + 1$
  else
    S.insertBack($S_2[j]$) {copy jth element of S_2 to end of S}
    $j \leftarrow j + 1$
while $i < S_1$.size() **do** {copy the remaining elements of S_1 to S}
  S.insertBack($S_1[i]$)
  $i \leftarrow i + 1$
while $j < S_2$.size() **do** {copy the remaining elements of S_2 to S}
  S.insertBack($S_2[j]$)
  $j \leftarrow j + 1$

Code Fragment 11.1: Algorithm for merging two sorted array-based sequences.



Merge-Sort and Recurrence Equations

- Recurrent relation or recurrence equation

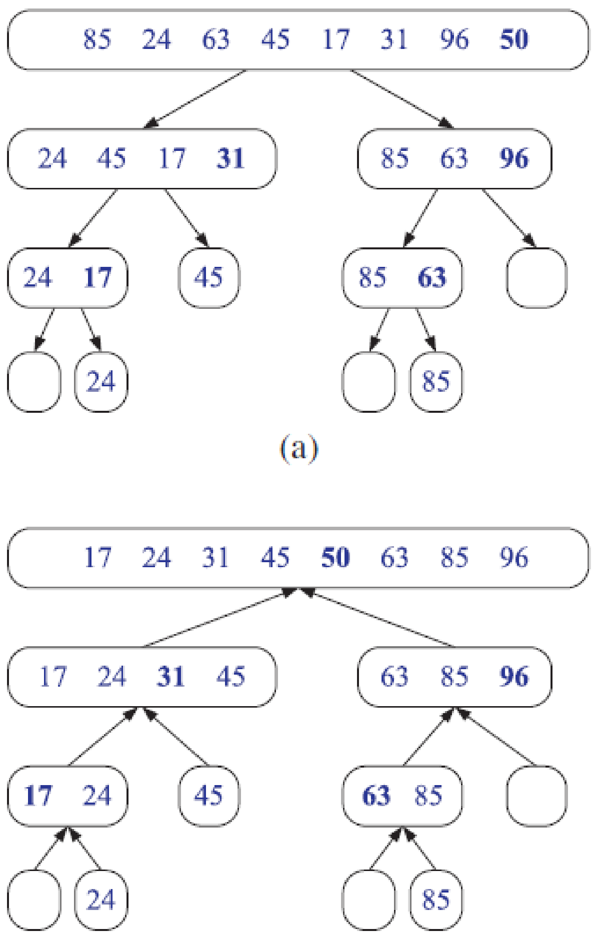
$$t(n) = \begin{cases} b & \text{if } n \le 1 \\ 2t(n/2) + cn & \text{otherwise.} \end{cases}$$

$$\begin{aligned} t(n) &= 2(2t(n/2^2) + (cn/2)) + cn \\ &= 2^2 t(n/2^2) + 2(cn/2) + cn \;=\; 2^2 t(n/2^2) + 2cn. \end{aligned}$$

$$t(n) = 2^i t(n/2^i) + icn.$$

$$\begin{aligned} t(n) &= 2^{\log n} t(n/2^{\log n}) + (\log n)cn \\ &= nt(1) + cn\log n \\ &= nb + cn\log n. \end{aligned}$$

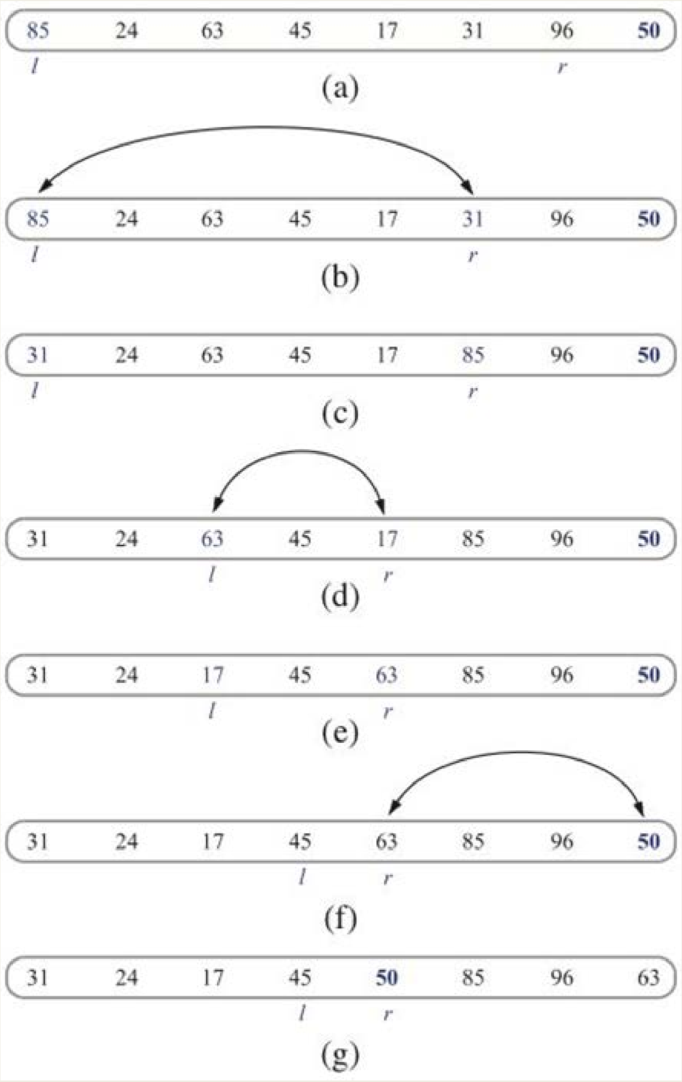
Quick-Sort



(a) (b)

```
Algorithm QuickSort(S):
  Input: A sequence S implemented as an array or linked list
  Output: The sequence S in sorted order
   if S.size() ≤ 1 then
     return          {S is already sorted in this case}
   p ← S.back().element()          {the pivot}
   Let L, E, and G be empty list-based sequences
   while !S.empty() do {scan S backwards, dividing it into L, E, and G}
     if S.back().element() < p then
       L.insertBack(S.eraseBack())
     else if S.back().element() = p then
       E.insertBack(S.eraseBack())
     else {the last element in S is greater than p}
       G.insertBack(S.eraseBack())
   QuickSort(L)          {Recur on the elements less than p}
   QuickSort(G)          {Recur on the elements greater than p}
   while !L.empty() do {copy back to S the sorted elements less than p}
     S.insertBack(L.eraseFront())
   while !E.empty() do {copy back to S the elements equal to p}
     S.insertBack(E.eraseFront())
   while !G.empty() do {copy back to S the sorted elements greater than p}
     S.insertBack(G.eraseFront())
   return          {S is now in sorted order}
```

In-place Quick-Sort




```
template <typename E, typename C>                         // quick-sort S
void quickSort(std::vector<E>& S, const C& less) {
  if (S.size() <= 1) return;                              // already sorted
  quickSortStep(S, 0, S.size()−1, less);                  // call sort utility
}

template <typename E, typename C>
void quickSortStep(std::vector<E>& S, int a, int b, const C& less) {
  if (a >= b) return;                                     // 0 or 1 left? done
  E pivot = S[b];                                         // select last as pivot
  int l = a;                                              // left edge
  int r = b − 1;                                          // right edge
  while (l <= r) {
    while (l <= r && !less(pivot, S[l])) l++;             // scan right till larger
    while (r >= l && !less(S[r], pivot)) r−−;             // scan left till smaller
    if (l < r)                                            // both elements found
      std::swap(S[l], S[r]);
  }                                                       // until indices cross
  std::swap(S[l], S[b]);                                  // store pivot at l
  quickSortStep(S, a, l−1, less);                         // recur on both sides
  quickSortStep(S, l+1, b, less);
}
```

Code Fragment 11.7: A coding of in-place quick-sort, assuming distinct elements.

Stable Sorting, Bucket-Sort, Radix-Sort

- Stable Sorting
  - *If entries of same keys order the same after sorting*
- Bucket-Sort (or Counting-Sort)
  - *O(n+N), where n is # of entries with integer keys in the range [0,N-1]*
- What if N ~ n^2?
  - *Bucket-Sort => O(n^2)*
  - *Radix-Sort => O(d(n+b)), b = 10, d = # of digits*
- Radix-Sort
  - *Sort digit-by-digit starting from least to most significant digit*
  - *Each digit is sorted by stable bucket sort.*

Algorithm bucketSort(S):
    Input: Sequence S of entries with integer keys in the range $[0, N-1]$
    Output: Sequence S sorted in nondecreasing order of the keys
    let B be an array of N sequences, each of which is initially empty
    for each entry e in S **do**
        $k \leftarrow e$.key()
        remove e from S and insert it at the end bucket (sequence) $B[k]$
    for $i \leftarrow 0$ to $N-1$ **do**
        for each entry e in sequence $B[i]$ **do**
            remove e from $B[i]$ and insert it at the end of S

Code Fragment 11.8: Bucket-sort.

170, 45, 75, 90, 802, 24, 2, 66
17**0**, 9**0**, 80**2**, **2**, 2**4**, 4**5**, 7**5**, 6**6**
8**0**2, 2, **2**4, **4**5, **6**6, 1**7**0, **7**5, **9**0
2, 24, 45, 66, 75, 90, **1**70, **8**02

Comparing Sorting Algorithms

- $O(n^2)$
  - *Insertion sort, Selection sort*
- $O(n \log n)$
  - *Merge sort*
    - No in-place algorithm, good for disk-based sorting, slower than quick and heap
  - *Quick sort*
    - Faster than merge and heap on average but bad on worst-case
  - *Heap sort*
    - Good on average and worst-case
- $O(n)$ for certain types of keys
  - *Bucket sort, Radix sort*

Sets

- Set vs. Bag vs. List

- Set
  - *insert(e)*
  - *find(e)*
  - *erase(e)*
  - *begin()*
  - *end()*
  - *union(B) : A <- A U B*
  - *intersect(B) : A <- A ∩ B*
  - *subtract(B) : A <- A - B*

- STL set class (ordered set)
  - *lower_bound(e)*
  - *upper_bound(e)*
  - *equal_range(e)*