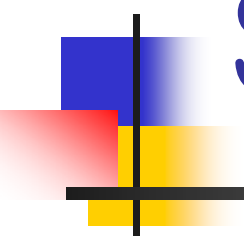


Sorting

Introduction to Data Structures

Kyuseok Shim

SoEECS, SNU.

Terminology

- List: a collection of records
  - ex) Student list of CS206 Data Structure
- Record: having one or more fields
  - ex) a student's data (name, age, sex, number …)
- Field: data
  - ex) name, age, …
- Key: distinguishing data
  - ex) name, number
- Sorting: making ordered list by the key
- Data type:

```
class Element
{
public
   int getKey() const { return key ;};
   void setKey(int k) { key = k;};
private:
   int key;
    // other fields
    .
    .
};
```

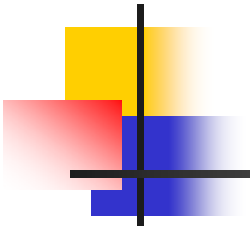
7.1 Motivation

- Sequential Search

```
template <class E, class K>
int SeqSearch (E *a, const int n, const K& k)
{ // Search a[1:n] from left to right. Return least i such that
  // the key of a[i] equals k. If there is no such i, return 0.
     int i;
     for(i=1; i<=n&&a[i]!=k; i++);
     if(i>n) return 0;
     return i;
}
```

Program 7.1: Sequential search

  - Analysis
    - worst case time : O(n)
- Two ways of storage and search
  - Sequential
  - Nonsequential



7.1 Motivation (Cont.)

- Sorting
  - Two uses of sorting
    - as an aid in searching
    - as a means for matching entries in lists
      - (ex. Comparing two lists)
  - If the list is sorted, the searching time could be reduced.
    - from : O(n) to : $O(\log_2 n)$
      - ex) Binary Search : $O(\log_2 n)$

Example 1

- Directly comparing the two unsorted lists

```
void Verify1(Element *ℓ1, Element *ℓ2, const int n, const int m)
{ // Compare two unordered lists l1 and l2 of size n and m, respectively.
    bool *marked = new bool[m+1];
    fill(marked+1, marked+m+1, false);

    for(i=0; i<=n; i++)
    {
        int j = SeqSearch(ℓ2, m, ℓ1[i]);
        if(j==0) cout << ℓ1[i] << " not in ℓ2 " << endl;
        else
        {
            if(!Compare(ℓ1[i], ℓ2[j])
                cout << "Discrepancy in "  << ℓ1[i] << endl;
            marked[j] = true; // mark ℓ2[j] as being seen
         }
    }
    for(i=1; i<=m; i++)
        if(!marked[i]) cout << ℓ2[i] << "not in ℓ1." << endl;
    delete [] marked;
}
```

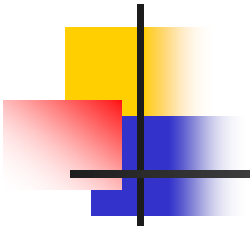
Program 7.2: Verifying two lists using a sequential search

Example 2

- Directly comparing the two sorted lists

```
void Verify2(Element *ℓ1, Element *ℓ2, const int n, const int m)
{ // Same task as Verify1. However, this time we fist sort ℓ1, ℓ2.
    Sort(ℓ1, n);  // sort into increasing order of key
    Sort(ℓ2, m);
    int i=0, j=0;
    while ( (i<=n) && (j<=m) )
        if (ℓ[i] < ℓ [j]) {
            cout << ℓ1[i] << " not in ℓ2" << endl;
            i++;
        }
        else if (ℓ[i] > ℓ [j]) {
            cout << ℓ2[j] << " not in ℓ1" << endl;
            j++;
        }
        else  { // equal keys
            if (!Compare(ℓ1[i], ℓ2[j]))
                cout << "Discrepancy in " << ℓ1[i] << endl;
            i++; j++;
        }
    if (i<=n) OutputRest(ℓ1, i, n, 1);          // output records i through n of ℓ1
    else if (j<=m) OutputRest(ℓ2, j, m, 2);     // output records j through m of ℓ2
}
```

Program 7.3: Fast verification of two lists



Example 1&2

- Time Complexity
  - Example 1
    - O(mn)
  - Example 2
    - $O(t_{sort}(n) + t_{sort}(m) + n + m)$

Another Example : Binary Search

- input : sorted list
- output : searched element

```
1 int BinarySearch (int *a, const int x, const int n)
2 // Search the sorted array a [0], ..., a [n-1] for x
3 {
4     for (int left = 0, right = n - 1; left <= right;) { // while more elements
5         int middle = (left + right)/2;
6         switch (compare (x, a[middle])) {
7             case '>': left = middle + 1; break; // x > a[middle]
8             case '<': right = middle - 1; break; // x < a[middle]
9             case '=': return middle; // x == a[middle]
10        } // end of switch
11    } // end of for
12    return -1; //  not found
13 } // end of BinarySearch
```

- Analysis : $O(\log_2 n)$

Sorting Terminology

- Record : $R_1, R_2, \cdots, R_n$
- List of records : $(R_1, R_2, \cdots, R_n)$
- Key value : K_i
- Ordering relation : $<$
- Transitive relation : $x < y$ and $y < z \Rightarrow x < z$
- Sorting Problem :
  - finding a permutation σ such that $K_{\sigma(i)} \le K_{\sigma(i+1)}$, $1 \le i \le n-1$
  - the desired ordering is $(R_{\sigma(1)}, R_{\sigma(2)}, \cdots, R_{\sigma(n)})$
- Stable sorting : σ_s
  - $K_{\sigma s(i)} \le K_{\sigma s(i+1)}$, $1 \le i \le n-1$
  - If $i < j$ and $K_i == K_j$, R_i precedes R_j in the sorted list
  - ex) input list : $6, 7, 3, 2_1, 2_2, 8$

    stable sorting : $2_1, 2_2, 3, 6, 7, 8$

    unstable sorting : $2_2, 2_1, 3, 6, 7, 8$

7.2 Insertion Sort

- Assume that ∃ a sorted list (R_1, R_2, ⋯, R_i).
- Add one element e.
- Artificial record R_0 with key K_0 = MININT(the smallest number)

```
template <class T>
void Insert(const T& e, T *a, int i)
{ // Insert e into the ordered sequence a[1:i] such that the
  // resulting sequence a[1:i+1] is also ordered.
  // The array a must have space allocated for at least i+2 elements.
     a[0] = e;
     while (e<a[i])
     {
          a[i+1] = a[i];
          i--;
     }
     a[i+1] = e;
}
```

Program 7.4: Insertion into a sorted list

7.2 Insertion Sort (Cont.)

```
template <class T>
void InsertionSort(T *a, const int n)
{ // Sort a[1:n] into nondecreasing order.
    for (int j=2; j<=n, j++) {
        T temp = a[j];
        Insert(temp, a, j-1);
    }
}
```

Program 7.5: Insertion Sort

7.2 Insertion Sort Example

- Example 7.1

j	[1]	[2]	[3]	[4]	[5]
–	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

7.2 Insertion Sort Analysis

- Time Complexity Analysis
  - Insert(e, list, i) $\Rightarrow$ i+1 comparisons
  - Insertionsort(list, n) $\Rightarrow$ n-1 times

  $O\left(\sum_{i=1}^{n-1}(i+1)\right) = O(n^2)$
- Useful when the given list is partially ordered.
- Stable sorting method
- Useful for small size sorting ($n \leq 20$)

7.3 Quick Sort

- One type of Insertion sort
- Pivot key K_i ,
  - If K_i is in position s(i),

    $K_j \le K_{s(i)}$ for $j < s(i)$

    $K_j \ge K_{s(i)}$ for $j > s(i)$

    $\Rightarrow$ two sublists $(R_1, \cdots, R_{s(i)-1})$

    $(R_{s(i)+1}, \cdots, R_n)$

- Some procedure in the sublists

Quick Sort Code

```
template <class T>
void QuickSort(T *a, const int left, const int right)
{ // Sort a[left:right] into nondecreasing order.
  // a[left] is arbitrarily chosen as the pivot. Variables i and j
  // are used to partition the subarray so that at any time a[m] ≤ pivot, m < i
  // and a[m] ≥ pivot, m > j. It is assumed that a[left] ≤ a[right + 1]
    if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
}
```

Program 7.6: Quick Sort

Quick Sort Example

- Example 7.3
  - n = 10
  - input list (26, 5, 37, 1, 61, 11, 59, 15, 48, 19)

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	*left*	*right*
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

Figure 7.1: Quick Sort example

Quick Sort Analysis

- Time complexity analysis
  - Worst case : $O(n^2)$
  - Optimal case : $T(n)$

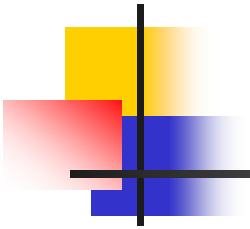
    $T(n) \le cn + 2T(n/2)$, for some constant c

    $\le cn + 2(cn/2 + 2T(n/4))$

    $\le 2cn + 4T(n/4)$

    :

    $\le cn\log_2 n + nT(1) = O(n\log n)$
- Unstable sorting
- Good(best) sorting method (average computing time is O(nlogn))



Quick Sort Average Time

- Lemma 7.1: Let $T_{avg}(n)$ be the expected time for function QuickSort to sort a list with records. Then there exists a constant k such that $T_{avg}(n) \leq knlog_e n$ for $n \geq 2$.

Quick Sort Average Time

- Proof ($T_{avg}(n) \le knlog_e n$ for $n \ge 2$)
  - We have

$$T_{avg}(n) \le cn + \frac{1}{n}\sum_{j=1}^{n}(T_{avg}(j-1) + T_{avg}(n-j)) = cn + \frac{2}{n}\sum_{j=0}^{n-1} T_{avg}(j) \quad (7.1)$$

  - We assume $T_{avg}(0) \le b$ and $T_{avg}(1) \le b$
  - Induction base: For n=2, $T_{avg}(2) \le 2c + 2b \le 2k \log_e 2.$
  - Induction hypothesis: Assume $T_{avg}(n) \le kn \log_e n$ for $1 \le n < m$
  - Induction step: From Eq. (7.1) and the induction hypothesis we have

$$T_{avg}(m) \le cm + \frac{4b}{m} + \frac{2}{m}\sum_{j=2}^{m-1} T_{avg}(j) \le cm + \frac{4b}{m} + \frac{2k}{m}\sum_{j=2}^{m-1} j \log_e j \quad (7.2)$$

  - Since $j\log_e j$ is an increasing function of j, Eq. (7.2) yields

$$T_{avg}(m) \le cm + \frac{4b}{m} + \frac{2k}{m}\sum_{j=2}^{m-1} j \log_e j \le cm + \frac{4b}{m} + \frac{2k}{m}\int_2^m x \log_e x dx = cm + \frac{4b}{m} + \frac{2k}{m}\left[\frac{m^2 \log_e m}{2} - \frac{m^2}{4}\right]$$

$$= cm + \frac{4b}{m} + km \log_e m - \frac{km}{2} \le km \log_e m \text{ , for } m \ge 2$$

7.4 How fast can we sort ?

- Worst : $O(n^2)$
- Best : $O(n\log_2 n)$
- Decision tree : describing sorting process
  - vertex – key comparison
  - branch – result

7.4 How fast can we sort ?

- Example 7.4
  - input (R_1, R_2, R_3) $\Rightarrow$ root is [1, 2, 3]
  - maximum depth of tree is 3

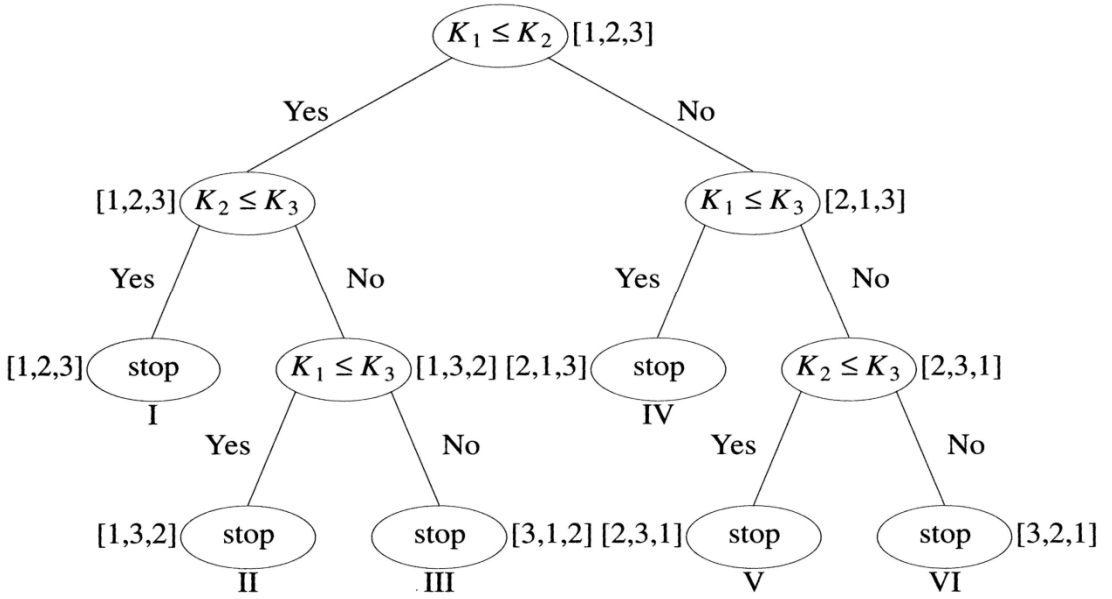
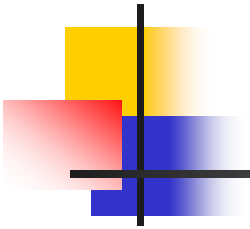


Figure 7.2: Decision tree for Insertion Sort



7.4 How fast can we sort ?

- Theorem 7.1
  - Any decision tree that sorts n distinct elements has a height of at least $\log_2(n!) + 1$
- Proof
  - When sorting n elements, there are n! different possible results. Thus, every decision tree for sorting must have at least n! leaves. But a decision tree is also a binary tree, which can have at most 2^{k-1} leaves if its height is k. Therefore, the height must be at least $\log_2 n!+1$

7.4 How fast can we sort ?

- Corollary
  - Any algorithm that sorts only by comparisons must have a worst case computing time of Ω(nlogn)
- Proof
  - We must show that for every decision tree with n! leaves, there is a path of length cnlog2n, where c is a constant. By the theorem, there is a path of length log2n!. Now

$$n! = n(n-1)(n-2)\cdots(3)(2)(1) \geq (n/2)^{n/2}$$

So, $\log_2 n! \geq (n/2)\log_2(n/2) = \Omega(n\log n)$.

7.5 Merge Sort

- 7.5.1 Merging
  - Merge two sorted lists to a single sorted list.

    (initList$_\ell$, ⋯, initList$_m$) (initList$_{m+1}$, ⋯, initList$_n$)

    ⇒ (mergeList$_\ell$, ⋯, mergeList$_n$)

  - Analysis

    O(n−ℓ+1) ⇒ O(n)

  - Example

    (3, 4, 8, 9) (5, 7, 10, 11)

    ⇒ (3, 4, 5, 7, 8, 9, 10, 11)

  - Stable sorting

Merge

```
template <class T>
void Merge(T *initList, T *mergedList, const int ℓ, const int m, const int n)
{ // initList [ℓ :m] and initList[m+1:n] are sorted lists. They are merged to obtain
  // the sorted list mergedList [ℓ :n]
    for (int i1 = ℓ, iResult = ℓ, i2 = m+1; // i1, i2, and iResult are list positions
           i1 <= m && i2 <= n; // neither input list if exhausted
           iResult++)
        if (initList[i1] <= initList[i2])
        {
            mergedList[iResult] = initList[i1];
            i1++;
         }
        else
        {
            mergedList[iResult] = initList[i2];
            i2++;
         }
     // copy remaining records, if any, of first list
    copy(initList+i1, initList+m+1, mergedList+iResult);
    // copy remaining records, if any, of second list
    copy(initList+i2, initList+n+1, mergedList+iResult);
}
```

Program 7.7: Merging two sorted lists

7.5.2 Iterative Merge Sort

- We assume that the input is n sorted lists. But each list is of length 1.
- These lists are merged by pairs to obtain n/2 lists.

```
template <class T>
void MergePass(T* initList, T* resultList, const int n, const int s)
{ // Adjacent pairs of sublists of size s are merged from
  // initList to resultList. n is the number of records in initList.

    for (int i=1; // i is first position in first of the sublists being merged
            i <= n-2*s+1; // enough elements for two sublists of length s?
            i += 2*s)
        Merge(initList, resultList, i, i + s - 1, i + 2 * s - 1);

    // merge remaining list of size < 2*s
    if( (i+s-1) < n) Merge(initList, resultList, i, i + s - 1, n);
    else copy(initList + i, initList + n + 1, resultList + i);
}
```

Program 7.8: Merge pass

7.5.2 Iterative Merge Sort

```
template <class T>
void MergeSort(T *a, const int n)
{ // Sort a[1:n] into nondecreasing order.

    T* tempList = new T[n+1];
    // ℓ is the length of the sublist currently being merged
    for (int  ℓ=1; ℓ<n; ℓ *= 2)
    {
        MergePass(a, tempList, n, ℓ);
        ℓ *= 2;
        MergePass(tempList, a, n, ℓ); // interchange role of a and tempList
    }
    delete [] tempList;
}
```

Program 7.9: Merge Sort

Iterative Merge Sort example

- Example 7.5
  - input list (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)

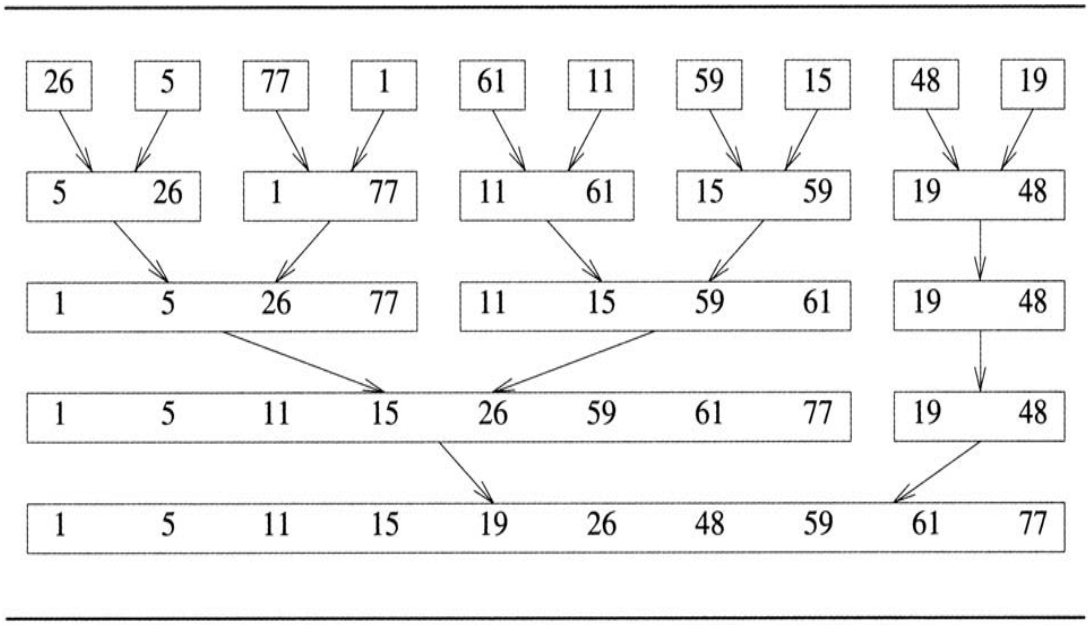
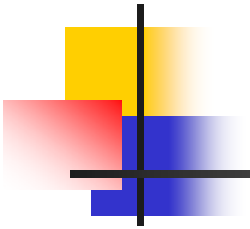


Figure 7.4: Merge tree



Iterative Merge Sort analysis

- Analysis :
  - merge-pass : O(n)
  - number of merge passes : $O(\log_2 n)$

    $\Rightarrow O(n\log_2 n)$
- Stable sorting

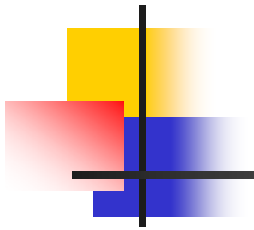
7.5.3. Recursive Merge Sort

- Divide the list into two sublists(left, right)
- Physical storage space is not changed.

```
template <class T>
int rMergeSort(T *a, int* link, const int left, const int right)
{ // a[left:right] is to be sorted. link[i] is initially 0 for all i.
  // rMergeSort returns the index of the first element in the sorted chain.


    if (left >= right) return left;
    int mid = (left + right) / 2;
    return ListMerge(a, link,
                     rMergeSort(a, link, left, mid),          // sort left half
                     rMergeSort(a, link, mid + 1, right) );   // sort right half
}
```

Program 7.10: Recursive Merge Sort



```
template <class T>
int ListMerge(T* a, int* link, const int start1, const int start2)
{ // The sorted chains beginning at start1 and start2, respectively, are merged.
  // link [0] is used as a temporary header. Return start of merged chain.
    int iResult = 0; // last record of result chain
    for (int i1 = start1, i2 = start2; i1 && i2; )
        if (a[i1] <= a[i2]) {
            link[iResult] = i1;
            iResult = i1; i1 = link[i1];
        }
        else {
            link[iResult] = i2;
            iResult = i2; i2 = link[i2];
        }

    // attach remaining records to result chain
    if(i1 == 0) link[iResult] = i2;
    else link[iResult] = i1;
    return link[0];
}
```

Program 7.11: Merging sorted chains

Variation
-- Natural Merge Sort

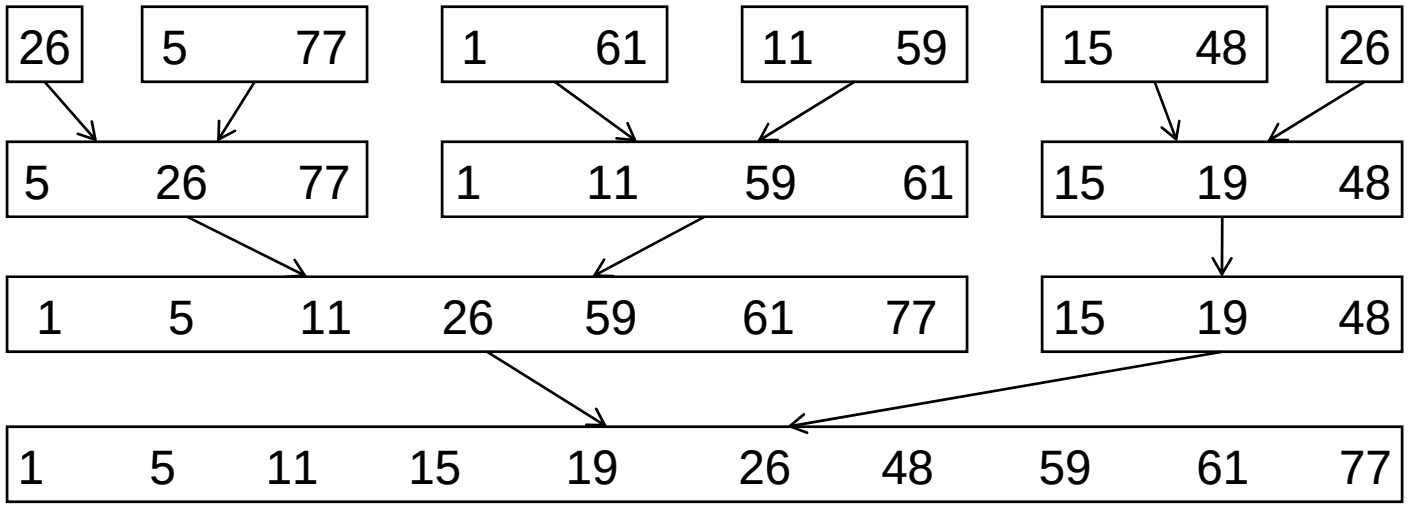
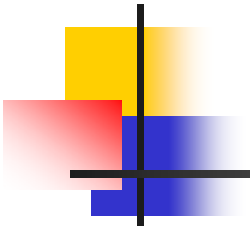


Figure 7.6: Natural Merge Sort



7.6 Heap Sort

- Save storage space
- Use the max heap structure in Chap. 5
- Store n element in the heap, and extract one at a time
- Adjust after the extract
- Store the extracted element in the last node

Adjusting a Max Heap

```
template <class T>
void Adjust(T *a, const int root, const int n)
{ // Adjust binary tree with root to satisfy heap property. The left and right
  // subtrees of root already satisfy the heap property. No node index is > n.

    T e = a[root];
    // find proper place for e
    for (int j = 2*root; j <= n; j *= 2) {
        if (j<n && a[j] < a[j+1]) j++; // j is max child of its parent
        if (e >= a[j]) break; // e may be inserted as parent of j
        a[j/2] = a[j]; // move j th record up the tree
    }
    a[j/2] = e;
}
```

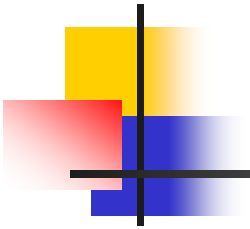
Program 7.13: Adjusting a max heap

Heap Sort

```
template <class T>
void HeapSort(T *a, const int n)
{ // Sort a[1:n]) into nondecreasing order.
     for (int i=n/2; i>= 1; i--) // heapify
          Adjust(a, i, n);

     for(i=n-1; i>=1; i--) // sort
     {
          swap(a[1], a[i+1]);        // swap first and last of current heap
          Adjust(a,1,i);             // heapify
     }
}
```

Program 7.14: Heap Sort



Heap Sort

- Analysis : O(nlogn)
  - Adjust : $\log_2 n$
  - Call : n

Example

- Example 7.7
  - input list (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)

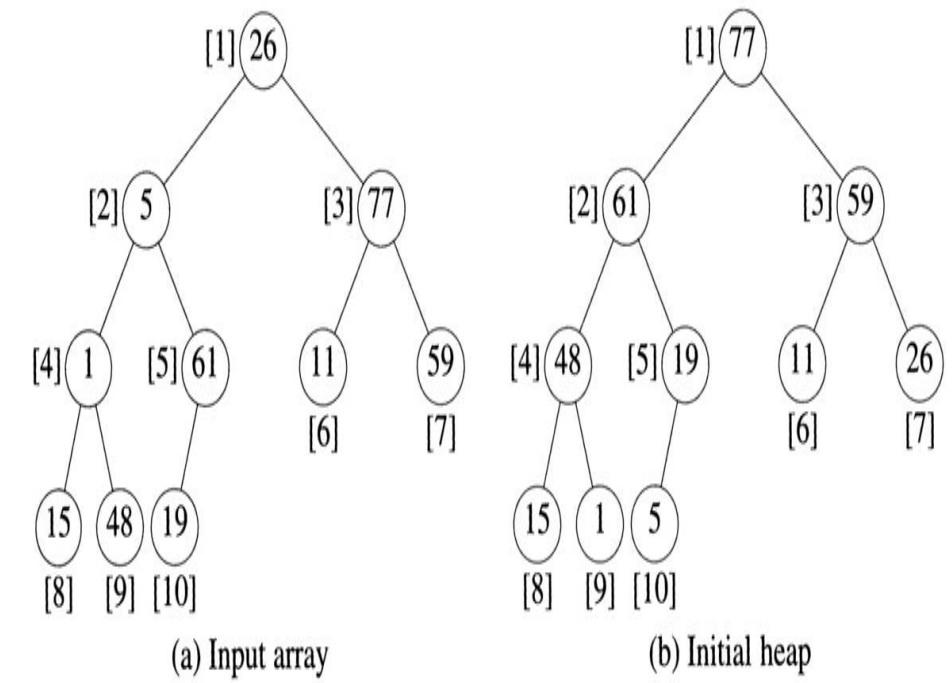


Figure 7.7: Array interpreted as a binary tree

Example

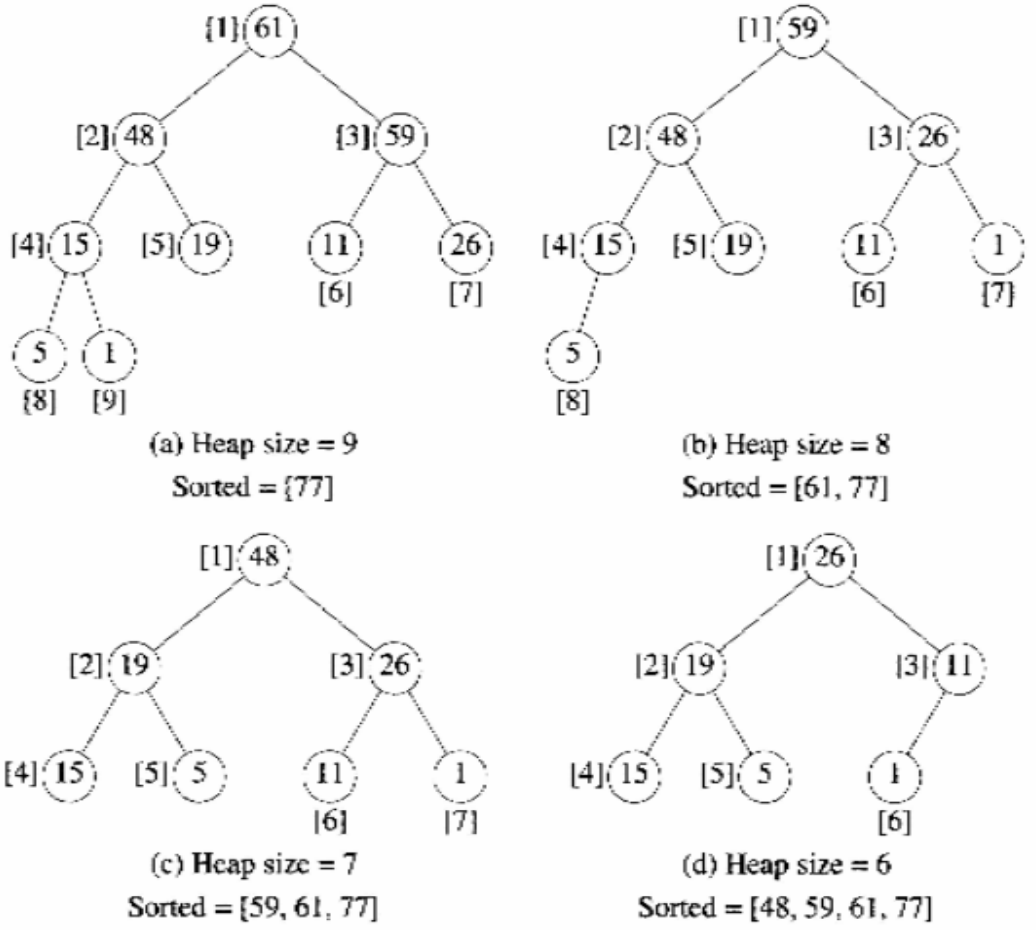


Figure 7.8: Heap Sort example (continued on next page)

Example

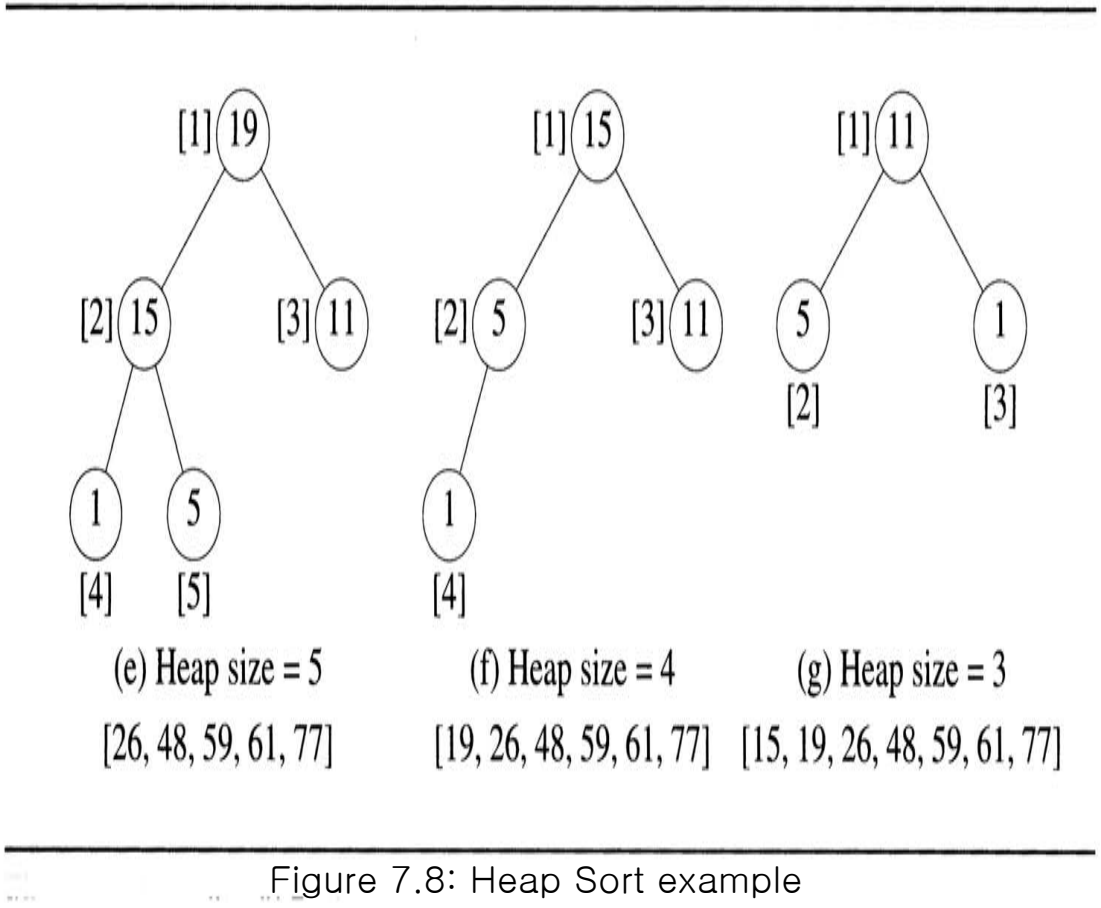


Figure 7.8: Heap Sort example

7.7 Sorting on Several Keys

- There are several keys K^1, K^2, …, K^r
  - K^1 is the most significant key(digit) : MSD
  - K^r is the least significant key(digit) : LSD
- $(R_1, \cdots, R_n)$ is sorted with respect to K^1, K^2, …, K^r iff $(K_i^1, \cdots, K_i^r) \leq (K_j^1, \cdots, K_j^r)$
- ex) a deck of cards

  K^1[suits] : ♣ < ♦ < ♥ < ♠

  K^2[values] : 2 < 3 < 4 …< 10 < J < Q < K < A

  Sorted deck :

  2 ♣, …, A♣, …, 2♠, …, A♠

LSD Radix Sort

```
template <class T>
int RadixSort(T *a, int *link, const int d, const int r, const int n)
{ // Sort a[1:n] using a d-digit radix-r sort. digit(a[i],j,r) returns the j th radix-r
  // digit (from the left) of a[i]'s key. Each digit is in the range is [0,r).
  // Sorting within a digit is done using a bin sort.

    int e[r], f[r]; // queue front and end pointers
    // create initial chain of records starting at first
    int first = 1;
    for(int i=1; i<n; i++) link[i] = i+1; // link into a chain
    link[n] = 0;

    for(i=d-1; i>=0; i--)
    { // sort on digit i
        fill(f, f+r, 0); // initialize bins to empty queues
        for(int current = first; current; current = link[current])
        { // put records into queues/bins
            int k = digit(a[current], i, r);
            if (f[k] == 0) f[k] = current;
            else link[e[k]] = current;
            e[k] = current;
         }
            ...
```

LSD Radix Sort (Cont.)

```
            ...
        for(j=0; !f[j]; j++); // find first nonempty queue/bin
        first = f[j];
        int last = e[j];
        for(int k=j+1; k<r; k++) // concatenate remaining queues
            if(f[k]) {
                link[last] = f[k];
                last = e[k];
            }
        link[last] = 0;
    }
    return first;
}
```

Program 7.15: LSD Radix Sort

- Analysis: RadixSort makes d passes, each pass takes O(n+r) time

  => O(d(n+r))

Radix Sort example

- Example 7.8

  10 number in [0,999], d=3, r=10

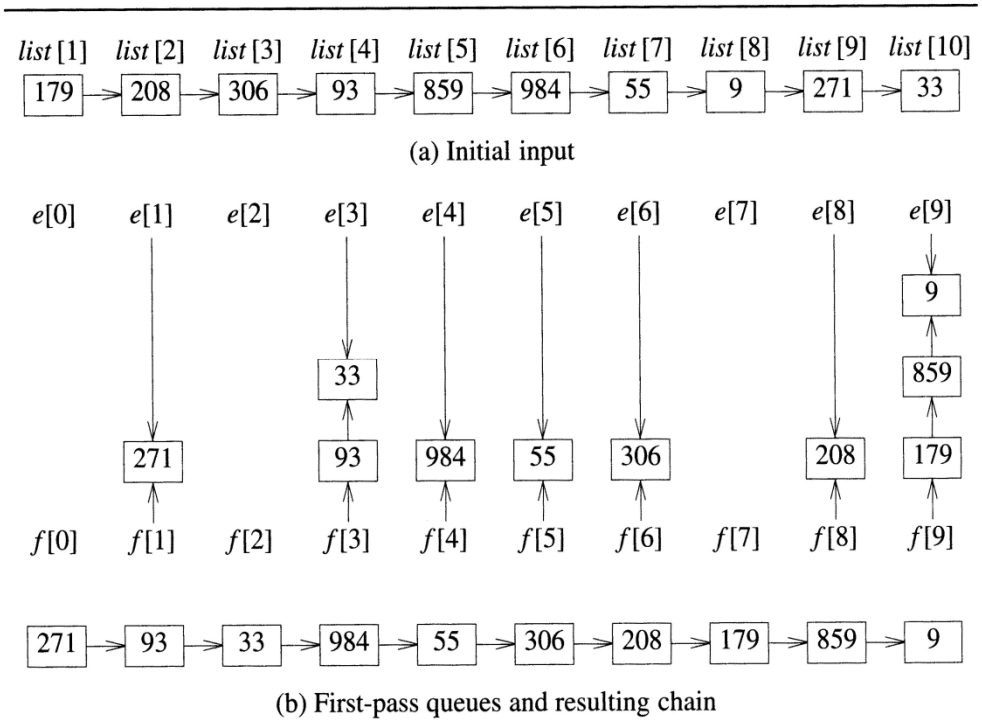
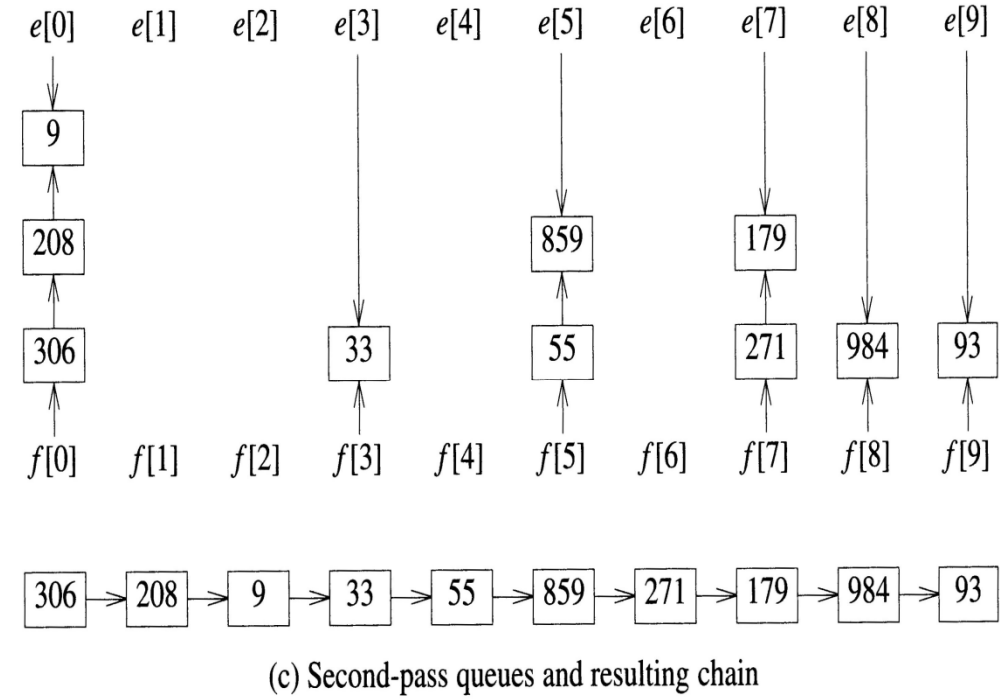


Figure 7.9: Radix Sort example (continued on next page)

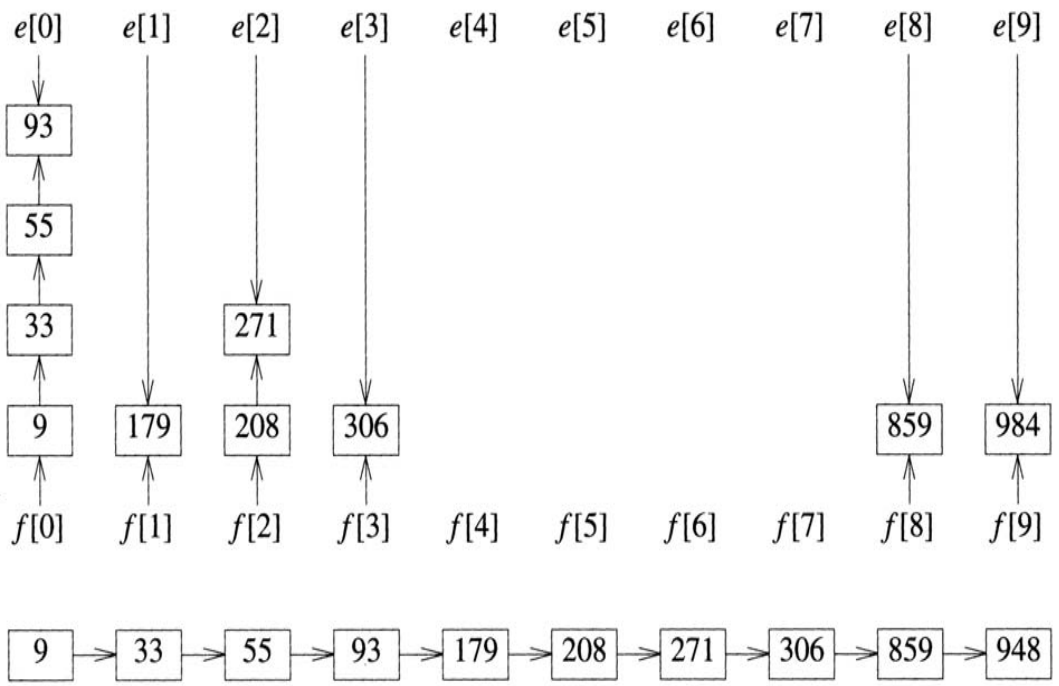
Radix Sort example (Cont.)



(c) Second-pass queues and resulting chain

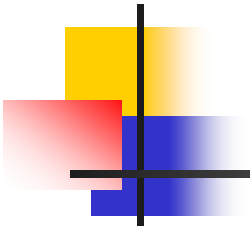
Figure 7.9: Radix Sort example (continued on next page)

Radix Sort example (Cont.)



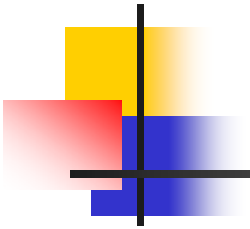
(d) Third-pass queues and resulting chain

Figure 7.9: Radix Sort example (continued on next page)



7.8 List and Table Sorts

- Physical arrangement ?
  - Radix (several keys), Recursive merge sort
    : no physical change (linked list)
  - Others : physical arrangement (array)
    - This requires excessive data movement
    - This tends to slow down the sorting process
- How to improve the ‘Others’?
  - First perform linked-list sort or table sort
  - Then physically rearrange the records
  - This rearranging can be accomplished in linear time using some additional space



List Sort

- Two cases of List Sort
  - doubly linked list
    - list[i].linka
    - list[i].linkb
  - singly linked list
    - list[i].link

Rearranging records using linked list

- At the end of linked list sort, the pointer first points to the first record
- We begin by interchanging records R_1 and R_{first}
  - Now the record in the position R_1 has the smallest key
- If first ≠ 1, then there is some record in the list whose link field is 1
- If we could change this link field to indicate the new position of the record previously at position 1, then we would left with records R_2, ⋯, R_n linked together in nondecreasing order
- Repeating the above process, after n−1 iterations, result in the desired rearrangement

Rearranging records using linked list : Doubly linked case

```
template <class T>
void List1(T *a, int *linka, const int n, int first)
{ // Rearrange the sorted chain beginning at first so that the records a[1:n]
  // are in sorted order.

    int *linkb = new int[n]; // array for backward links
    int prev = 0;
    for (int current = first; current; current = linka[current])
    { // convert chain into a doubly linked list
        linkb[current] = prev;
        prev = current;
    }
    for (int i=1; i<n; i++)          // move a[first] to position i while
    {                                // maintaining the list
        if(first != i) {
            if(linka[i]) linkb[linka[i]] = first;
            linka[linkb[i]] = first;
            swap(a[first], a[i]);
            swap(linka[first], linka[i]);
            swap(linkb[first], linkb[i]);
        }
        first = linka[i];
    }
}
```

Program 7.16: Rearranging records using a doubly linked list

Rearranging records using linked list : Doubly linked case (Cont.)

- Analysis
  - The time required to convert the chain first into a doubly linked list is O(n)
  - The second for loop is iterated n-1 times. In each iteration, at most two records are interchanged(3 record moves). if each record is m words long , the cost per record swap is O(m)
  - Total time is therefore O(mn)

Example 7.9

- input list (26, 5, 7, 1, 61, 11, 59, 15, 48, 19)

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	26	5	77	1	61	11	59	15	48	19
linka	9	6	0	2	3	8	5	10	7	1

(a) Linked list following a list sort, *first* = 4

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	26	5	77	1	61	11	59	15	48	19
linka	9	6	0	2	3	8	5	10	7	1
linkb	10	4	5	0	7	2	9	6	1	8

(b) Corresponding doubly linked list, *first* = 4

Figure 7.10: Sorted linked lists

Example for List1 (Program 7.16)

i	**R_1**	R_2	R_3	**R_4**	R_5	R_6	R_7	R_8	R_9	R_{10}
key	**1**	5	77	**26**	61	11	59	15	48	19
linka	**2**	6	0	**9**	3	8	5	10	7	4
linkb	**0**	4	5	**10**	7	2	9	6	4	8

(a) Configuration after first iteration of the **for** loop of *List*1, *first* = 2

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_9	R_9	R_{10}
key	1	5	77	26	61	11	59	15	48	19
linka	2	6	0	9	3	8	5	10	7	4
linkb	0	4	5	10	7	2	9	6	4	8

(b) Configuration after second iteration, *first* = 6

i	R_1	R_2	**R_3**	R_4	R_5	**R_6**	R_7	R_8	R_9	R_{10}
key	1	5	**11**	26	61	**77**	59	15	48	19
linka	2	6	**8**	9	6	**0**	5	10	7	4
linkb	0	4	**2**	10	7	**5**	9	6	4	8

(c) Configuration after third iteration, *first* = 8

i	R_1	R_2	R_3	**R_4**	R_5	R_6	R_7	**R_8**	R_9	R_{10}
key	1	5	11	**15**	61	77	59	**26**	48	19
linka	2	6	8	**10**	6	0	5	**9**	7	8
linkb	0	4	2	**6**	7	5	9	**10**	8	8

(d) Configuration after fourth iteration, *first* = 10

Figure 7.11: Example for List1 (Program 7.16)

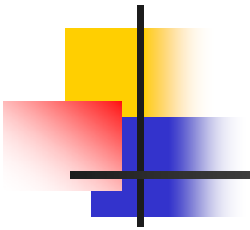
Rearranging records using linked list : Singly linked list

- When R_p is exchanged with R_i, the link of R_i is set to p
- first ≥ i

```
template <class T>
void List2(T *a, int *link, const int n, int first)
{ // Same function as List1 except that a second link array linkb is not required.

    for( int i=1; i<n; i++)
    { // Find correct record for i th position. Its index is ≥ i as
      // records in positions 1, 2, ···, i-1 are already correctly positioned.
        while(first < i) first = link[first];
        int q = link[first]; // a[q] is next record in sorted order
        if (first != i)
        { // a[first] has i th smallest key. Move record to i the position.
          // Also set link from old position of a[i] to new one.
            swap(a[i], a[first]);
            link[first] = link [i];
            link[i] = first;
        }
        first = q;
    }
}
```

Program 7.17: Rearranging records using only one link field



Rearranging records using linked list : Singly linked list (Cont.)

- Analysis : O(n)

Example for List2 (Program 7.17)

i	**R_1**	R_2	R_3	**R_4**	R_5	R_6	R_7	R_8	R_9	R_{10}
key	**1**	5	77	**26**	61	11	59	15	48	19
link	**4**	6	0	**9**	3	8	5	10	7	1

(a) Configuration after first iteration of the **for** loop of *List2*, *first* = 2

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	77	26	61	11	59	15	48	19
link	4	6	0	9	3	8	5	10	7	1

(b) Configuration after second iteration, *first* = 6

i	R_1	R_2	**R_3**	R_4	R_5	**R_6**	R_7	R_8	R_9	R_{10}
key	1	5	**11**	26	61	**77**	59	15	48	19
link	4	6	**6**	9	3	**0**	5	10	7	1

(c) Configuration after third iteration, *first* = 8

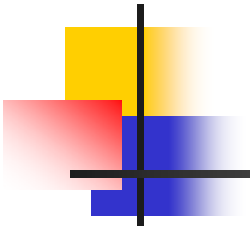
i	R_1	R_2	R_3	**R_4**	R_5	R_6	R_7	**R_8**	R_9	R_{10}
key	1	5	11	**15**	61	77	59	**26**	48	19
link	4	6	6	**8**	3	0	5	**9**	7	1

(d) Configuration after fourth iteration, *first* = 10

i	R_1	R_2	R_3	R_4	**R_5**	R_6	R_7	R_8	R_9	**R_{10}**
key	1	5	11	15	**19**	77	59	26	48	**61**
link	4	6	6	8	**10**	0	5	9	7	**3**

(e) Configuration after fifth iteration, *first* = 1

Figure 7.12: Example for List2 (Program 7.17)



List1 vs List2

- List1 is inferior to List2 in both space and time
  - List1 uses two link fields

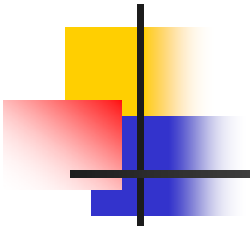


Table Sort

- A auxiliary table t
  - One entry per record
  - The entries serve as an indirect reference to the records
- Sorting
  - At the start of the sort
    - t[i] = i, $1 \le i \le n$
  - When the sorting function requires a swap of a[i] and a[j]
    - Swap t[i] and t[j], instead of a[i] and a[j]
  - At the end of the sort
    - $a[t[1]] \le a[t[2]] \le \cdots \le a[t[n]]$

Table Sort (Cont.)

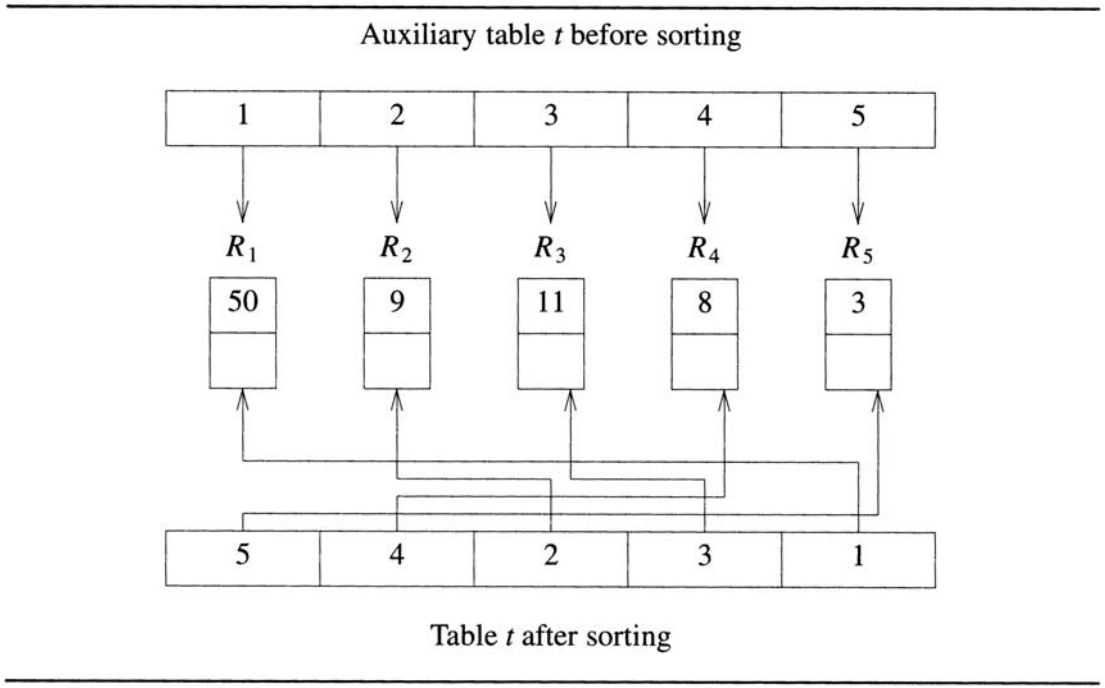
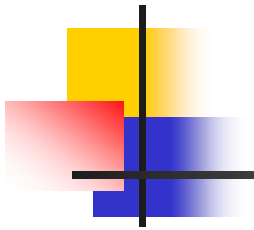


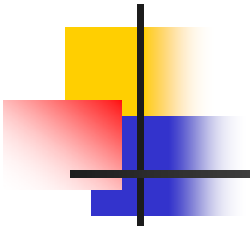
Figure 7.13: Table Sort

- t[1]=5, t[2]=4, t[3]=2, t[4]=3, t[5]=1
- a[t[1]] ≤ a[t[2]] ≤ a[t[3]] ≤ a[t[4]] ≤ a[t[5]]

Table Sort (Cont.)



- Every permutation is made up of disjoint cycles
- The cycle for any element i is made up of
  - i, $t[i]$, $t^2[i]$, ⋯, $t^k[i]$, where $t^j[i]=t[t^{j-1}[i]]$, $t^0[i]=i$, and $t^k[i]=i$
- Figure 7.13 has two cycles
  - R1, R5
  - R2, R4, R2



Rearranging records using table

- Physical rearrangement of table
  - Find cycles of records
  - Interchange records in a cycle using a temporary space

Rearranging records using table (Cont.)

```
template <class T>
void Table(T *a, const int n, int *t)
{ // Rearrange a[1:n] to correspond to the sequence a[t[1]], ···, a[t[n]], n ≥ 1.

    for(int i=1; i<n; i++)
        if(t[i] != i) { // there is a non-trivial cycle starting at i
            T p = a[i];
            int j = i;
            do {
                int k = t[j]; a[j] = a[k]; t[j]= j;
                j = k;
            } while (t[j] != i);
            a[j] = p; // j is position for record p
            t[j] = j;
        }
}
```

Program 7.18: Table Sort

- Analysis : O(mn) when the record length is m.
  But O(n) if m is fixed

Example 7.11

Cycle 1 : R_1, R_3, R_8, R_6, R_1

Cycle 2 : R_4, R_5, R_7, R_4

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
key	35	14	12	42	26	50	31	18
t	3	2	8	5	7	1	4	6

(a) Initial configuration

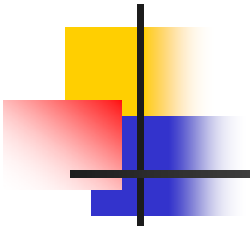
key	12	14	18	42	26	35	31	50
t	1	2	3	5	7	6	4	8

(b) Configuration after rearrangement of first cycle

key	12	14	18	26	31	35	42	50
t	1	2	3	4	5	6	7	8

(c) Configuration after rearrangement of second cycle

Figure 7.14: Table Sort example



7.9 Summary of Internal Sorting

Method	Worst	Average
Insertion Sort	n^2	n^2
Heap Sort	nlogn	nlogn
Merge Sort	nlogn	nlogn
Quick Sort	n^2	nlogn

Figure 7.15: Comparison of sort methods

Average times for sort methods

n	Insert	Heap	Merge	Quick
0	0.000	0.000	0.000	0.000
50	0.004	0.009	0.008	0.006
100	0.011	0.019	0.017	0.013
200	0.033	0.042	0.037	0.029
300	0.067	0.066	0.059	0.045
400	0.117	0.090	0.079	0.061
500	0.179	0.116	0.100	0.079
1000	0.662	0.245	0.213	0.169
2000	2.439	0.519	0.459	0.358
3000	5.390	0.809	0.721	0.560
4000	9.530	1.105	0.972	0.761
5000	15.935	1.410	1.271	0.970

Figure 7.16: Average times for sort methods (milliseconds)

Plot

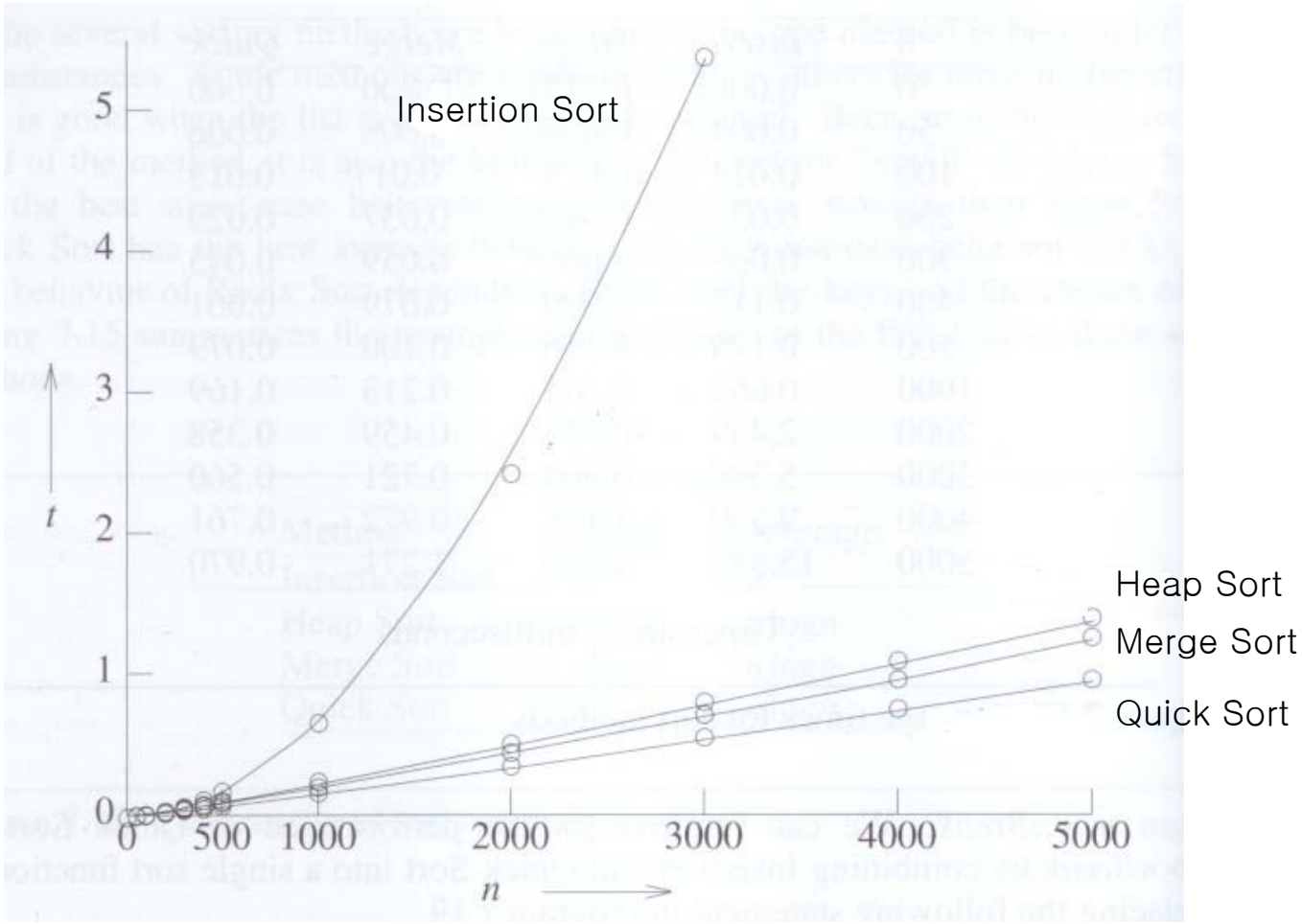
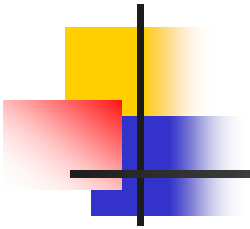
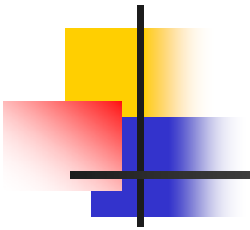


Figure 7.17: Plot of average times(miliseconds)



7.10 External Sorting

- When the lists to be sorted are so large
  - The whole list cannot be contained in the internal memory
  - Making an internal sort impossible
- The most popular method for sorting on external storage devices is Merge Sort
- External Merge Sort
  - Segments of the input list are sorted using a good internal sort method
  - The sorted segments are merged together until only one run is left



7.10 External Sorting (Cont.)

- Block
  - The unit of data that is read or written to a disk at one time
  - Consists of several records
- Three factors contributing to the read/write time
  - Seek time: time taken to position the read/write heads to the correct cylinder
  - Latency time: time until the right sector of the track is under the read/write head
  - Transmission time: time to transmit the block of data to/from the disk

Example 7.12

- lists: 4500 records
  internal memory capable of sorting: 750 records
  block length: 250 records

1. Internally sort three blocks at a time to obtain six runs R_1 to R_6
   - Heap Sort, Merge Sort, or Quick sort could be used
2. These six runs are written onto the scratch disk

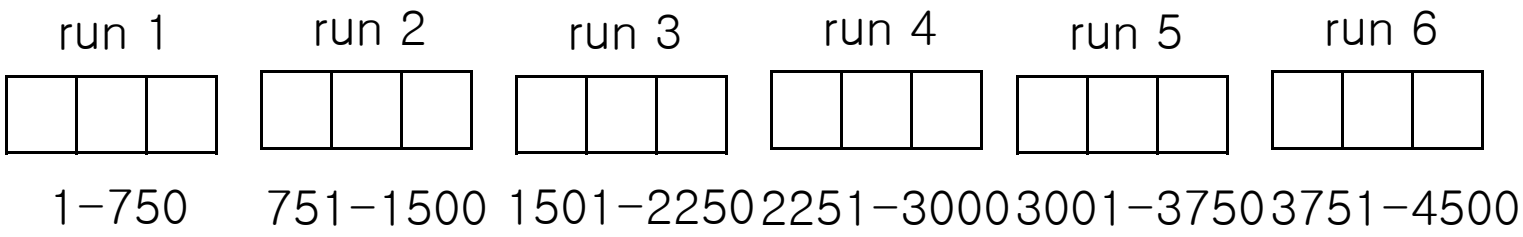
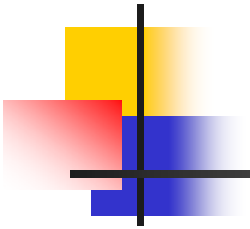


Figure 7.20: Blocked runs obtained after internal sorting



Example 7.12 (Cont.)

3. Set aside three blocks of internal memory
   - each capable of holding 250 records
   - Two for input buffers, one for an output buffer
4. Merge a pair of runs
   - First Read one block of each of two runs into input buffers
   - Blocks of runs are merged from the input buffers into the output buffer
   - When the output buffer gets full, it is written onto the disk
   - If an input buffer gets empty, it is refilled with another block from the same run
5. Merge a pair of runs generated in the previous step until only one run is left

Example 7.12 (Cont.)

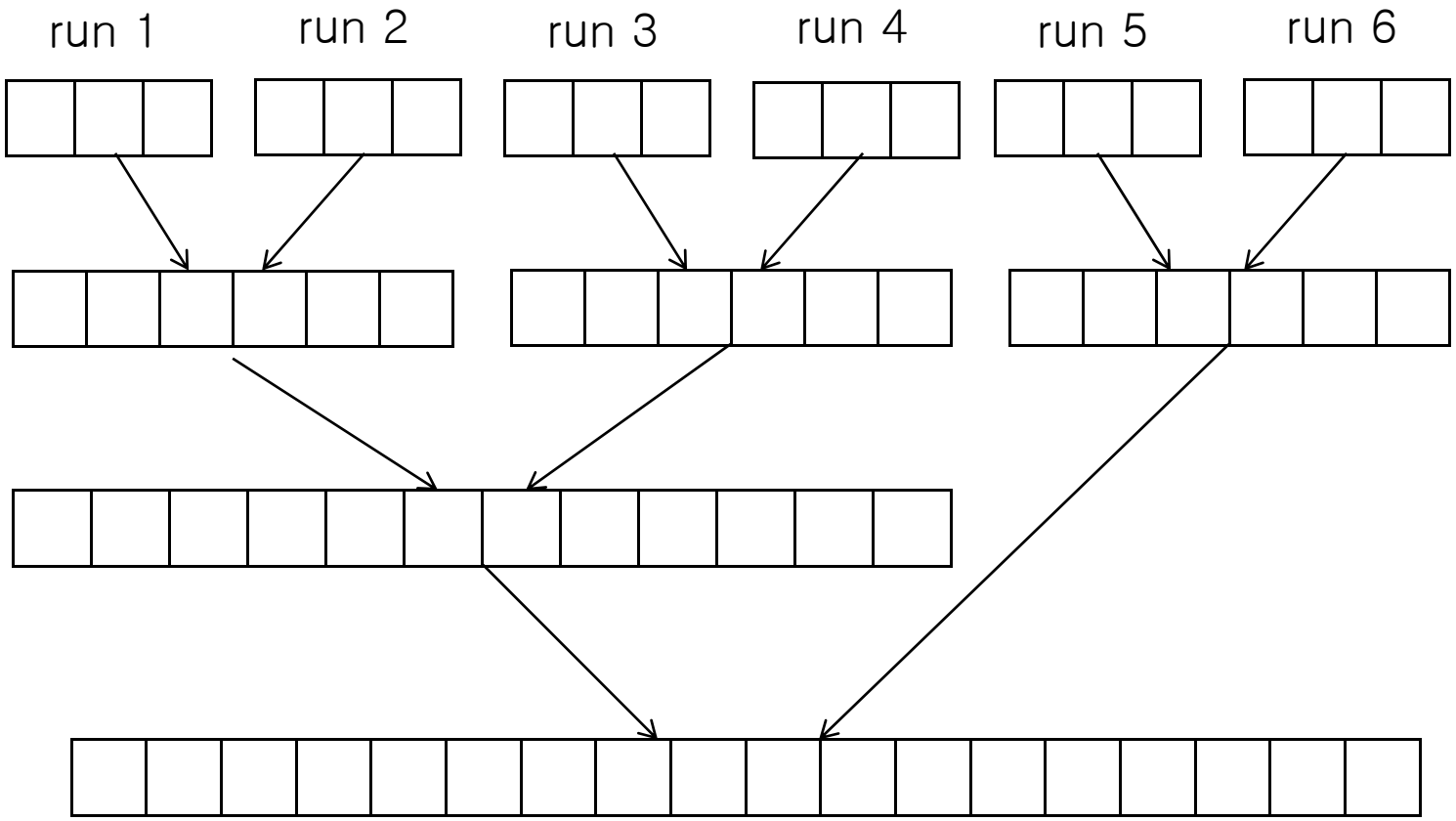


Figure 7.21: Merging the six runs

Time Complexity Analysis

- t_s = maximum seek time
- t_l = maximum latency time
- t_{rw} = time to read or write one block of 250 records
- t_{IO} = time to input or output one block = $t_s + t_l + t_{rw}$
- t_{IS} = time to internally sort 750 records
- $n*t_m$ = time to merge n records from input buffers to the output buffer

operation	time
(1) read 18 blocks of input, $18t_{IO}$, internally sort, $6t_{IS}$, write 18 blocks, $18t_{IO}$	$36t_{IO} + 6t_{IS}$
(2) merge runs 1 to 6 in pairs	$36t_{IO} + 4500t_m$
(2) merge two runs of 1500 records each, 12 blocks	$24t_{IO} + 3000t_m$
(4) merge one run of 3000 records with one run of 1500 records	$36t_{IO} + 4500t_m$
total time	$132t_{IO} + 12{,}000t_m + 6t_{IS}$

Figure 7.22: Computing times for disk sort example

k-Way Merging

- The number of merge passes can be reduced by using higher-order merge than two-way merge

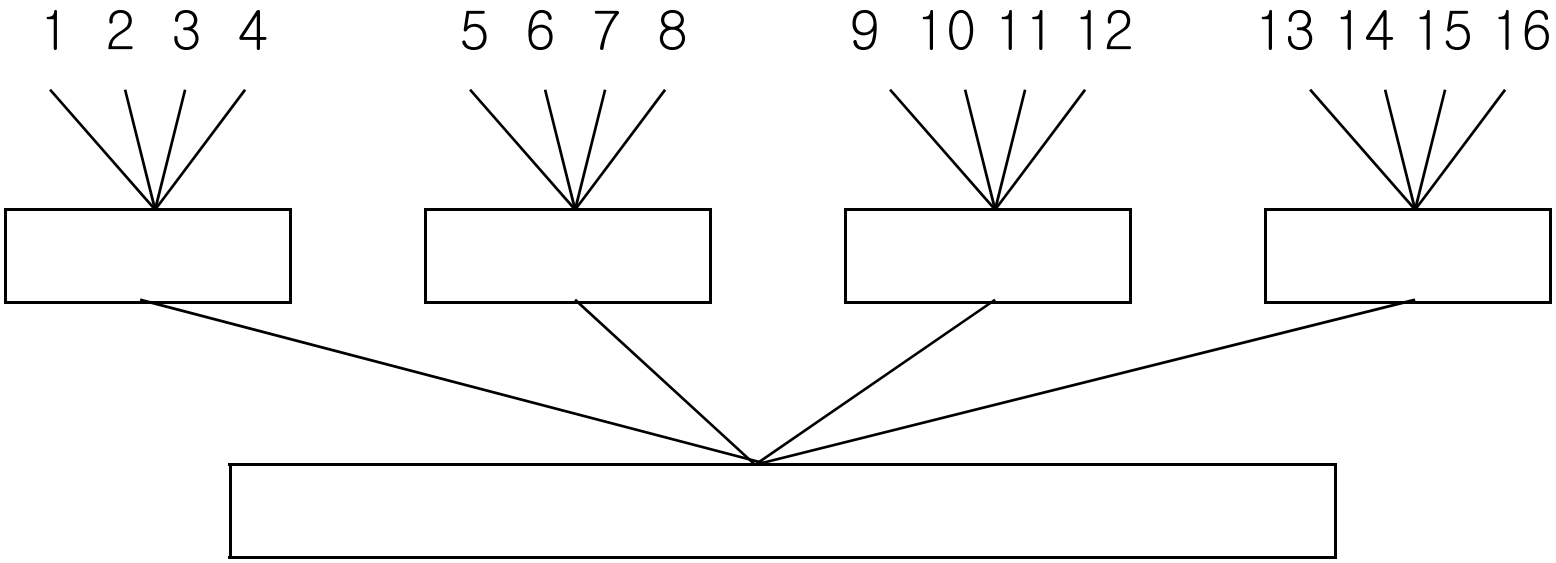
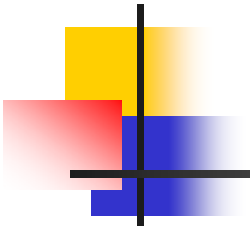


Figure 7.23: A four-way merge on 16 runs

k-Way Merging (Cont.)

- If we start with m runs, the two way merge tree will have $\lceil \log_2 m \rceil + 1$ levels
  - $\lceil \log_2 m \rceil$ passes
- k-way merge on m runs requires $\lceil \log_k m \rceil$ passes over the data
- k runs of size $s_1, s_2, s_3, \cdots, s_k$ can no longer be merged internally in $O(\sum_i s_i)$ time
  - In the most direct way, k-1 comparisons are needed to determine the next record to output $O((k-1)\sum_i s_i)$
  - total number of key comparisons is

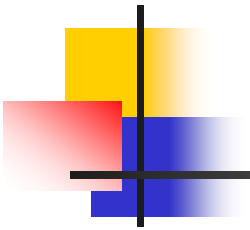
    $n(k-1)\log_k m = n(k-1)\log_2 m / \log_2 k$
- As k increases, the reduction in input/output time will be outweighed by the resulting increasing in CPU time
- At least, k + 1 buffers are needed



k-Way Merging (Cont.)

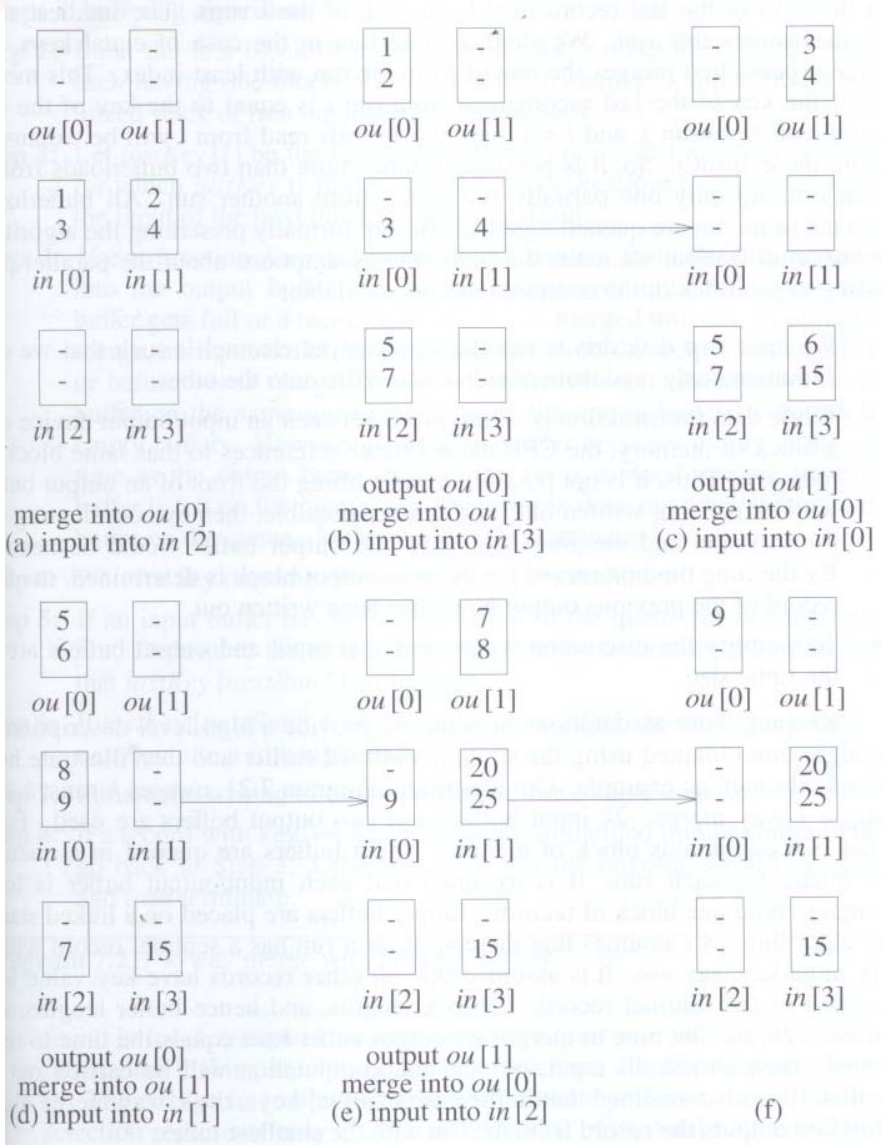
- For large k (say, k≥6), Loser tree may reduce the number of comparisons
  - Total time needed per level of the merge tree $O(n\log_2 k)$
  - The asymptotic internal processing time becomes $O(n\log_2 k\log_k m) = O(n\log_2 m)$
  - This is independent of k
  - Disk IO will be reduced
- The buffer size must be reduced as k increases
  - If the buffer size is reduced even smaller than block size, the number of block reading/writing will increase
- The optimal value for k depends on disk parameters and the amount of internal memory

7.10.3 Buffer Handling for Parallel Operation



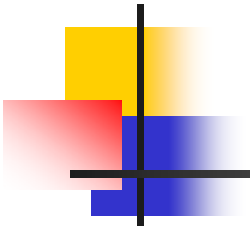
- While the output buffer is being written out, internal merging has to be halted
  - no place to collect the merged records
  - This can be overcome through the use of two output buffer
- With only k input buffers, internal merging will have to be held up whenever one of these input buffers becomes empty
  - This can be avoided if we have 2k input buffers
  - 2k input buffers have to be used cleverly to avoid a lack of input records from any one run

Example 7.13



- two-way merge
- ou[0] and ou[1] are output buffers
- in[0~3] are input buffers
- in[0] and in[2] are assigned to run 0
- in[1] and in[3] are assigned to run 1
- Assume times to input, output, and generate an output buffer are all the same

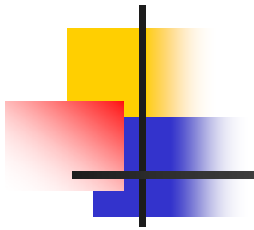
- run 0: 1, 3, 5, 7, 9
- run 1: 2, 4, 6, 15, 20, 25



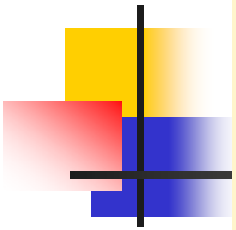
k-way merge with floating buffers

- An individual buffer may be assigned to any run depending upon need
- The run which will be empty first is the one from which the next buffer will be filled
- The remaining buffers will be filled on a priority basis
- Run-exhaustion prediction
  - Compare the keys of the last record read from each of the k runs
  - Use a looser tree

k-way merge with floating buffers (Cont.)



- 2k input buffers and two output buffers are used
- Input buffers are queued in k queues, one queue for each run
- Empty buffers are placed on a linked stack
- The end of each run has a sentinel records with a very large key, say $+\infty$
- If the time to merge one output buffer load equals the time to read a block, almost all input, output, and computation will be carried out in parallel



{ steps in buffering algorithm }

Step 1: Input the first block of each of the k runs, setting up k linked queues, each having one block of data. Put the remaining k input blocks into a linked stack of free input blocks. Set *ou* to 0.

Step 2: Let *lastKey[i]* be the last key input from run i. Let *nextRun* be the run for which *lastKey* is minimum. If *lastKey[nextRun]* $\neq +\infty$, then initiate the input of the next block from run *nextRun.*

Step 3: Use a function *Kwaymerge* to merge records from the k input queues into the output buffer *ou*. Merging continues until either the output buffer gets full or a records with key $+\infty$ is merged into *ou*. If, during this merge, an input buffer becomes empty before the output buffer gets full or before $+\infty$ is merged into *ou*, *the Kwaymerge* advances to the next buffer on the same queue and returns the empty buffer to the stack of empty buffers. However, if an input buffer becomes empty at the same time as the output buffer gets full or $+\infty$ is merged into *ou*, the empty buffer is left on the queue, and *Kwaymerge* does not advance to the next buffer on the queue. Rather, the merge terminates.

Step 4: Wait for any ongoing disk input/output to complete.

Step 5: If an input buffer has been read, add it to the queue for the appropriate run. Determine the next run to read from by determining *NextRun* such that *lastKey[nextRun]* is minimum.

Step 6: If *lastKey[nextRun]* $\neq +\infty$, then initiate reading the next block from run *nextRun* into a free input buffer.

Step 7: Initiate the writing of output buffer *ou*. Set *ou* to 1 – *ou*.

Step 8: If a record with key $+\infty$ has been not been merged into the output buffer, go back to Step 3. Otherwise, wait for the ongoing write to complete and then terminate.

Program 7.21: k-way merge with floating buffers

Example 7.14

Run 0	20 25	26 28	29 30	33 +∞
Run 1	23 29	34 36	38 60	70 +∞
Run 2	24 28	31 33	40 43	50 +∞

Figure 7.25: Three runs

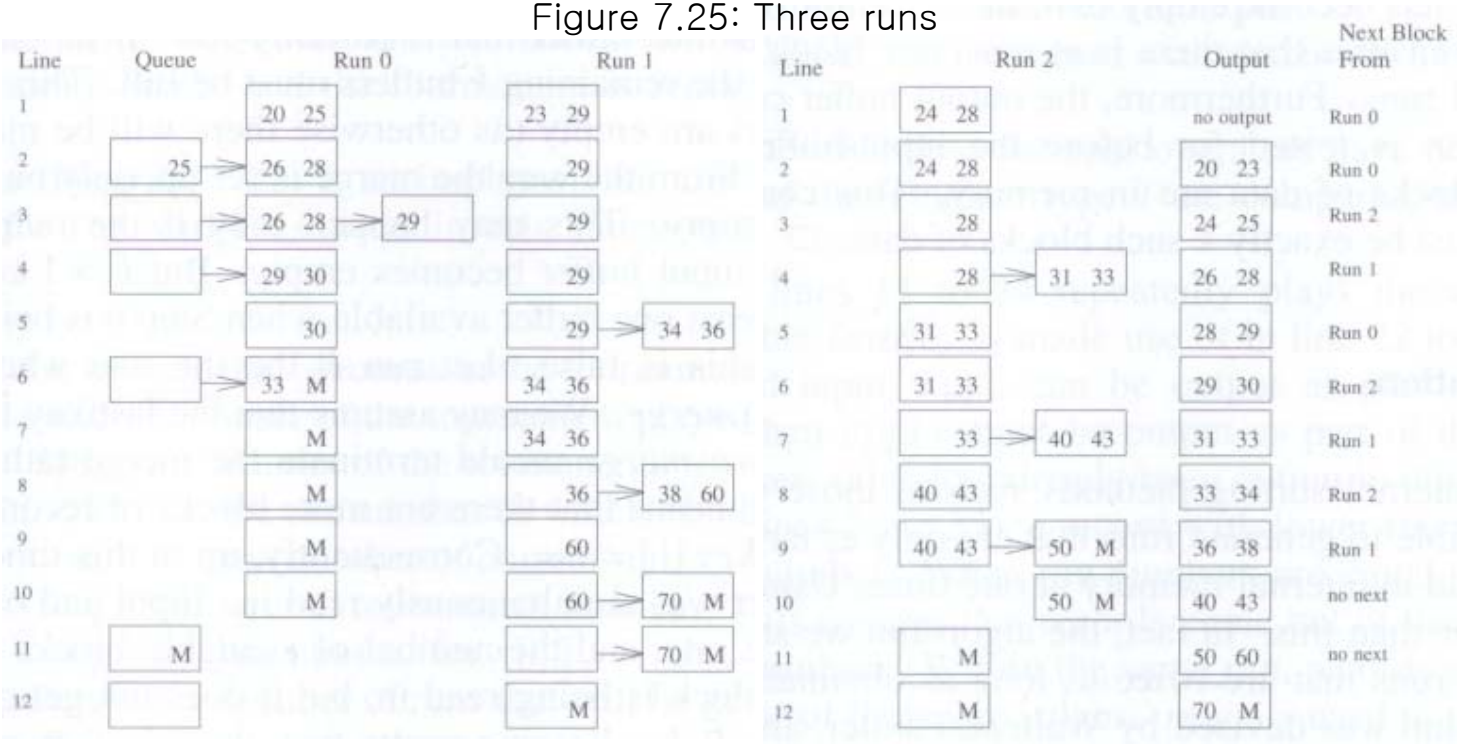


Figure 7.27 Buffering example

7.10.4 Run Generation

- The runs generated in this algorithm will be twice as long as obtainable by conventional methods(as large as the number of records that can be held in internal memory)
- This algorithm uses a loser tree
- The variables used in this algorithm
  - r[i], $0 \le i < k$ — the k records in the tournament tree
  - l[i], $1 \le i < k$ — loser of the tournament played at node i
  - l[0] — winner of the tournament
  - rn[i], $0 \le i < k$ — the run number to which r[i] belongs
  - rc — run number of current run
  - q — overall tournament winner
  - rq — run number for r[q]
  - rmax — number of runs that will be generated
  - lastRec — last record output
  - MAXREC — a record with maximum key possible

Run Generation using a loser tree : Brief Explanation

- The loop repeatedly plays the tournament outputting records (lines 11 to 34)
- The variable *lastKey* is made use of in line 22 to determine whether or not the new record input, r[q], can be output as part of the current run
  - if(key[q] < *lastKey*) then r[q] cannot be output as part of the current run *rc*
  - A record with larger key value has already been output in this run
- When the tree is being readjusted (line 27 to 33)
  - A record with lower run number winds over one with a higher run number
  - When run numbers are equal, the record with lower key value wins
- *rmax* is used to terminate the function. In line 19, when we run out of input, a record with num number *rmax*+1 is introduced. When this record is ready for output, the function terminates from line 14

Run Generation using a loser tree : Code

```
template <class T>
1.   void Runs(T *r)
2.   {
3.        r = new T[k];
4.        int *rn = new int[k], *l = new int[k];
5.        for (int i=0; i<k; i++) {  // input records
6.             InputRecord(r[i]); rn[i] = 1;
7.        }
8.        InitializeLoserTree();
9.        T q = l[0];  // tournament winner
10.       int rq = 1, rc = 1, rmax = 1; T lastRec = MAXREC;
11.       while(1) { // output runs
12.            if(rq != rc) {  // end of run
13.                 output end of run marker;
14.                 if (rq > rmax) return;
15.                 else rc = rq;
16.            }
17.            WirteRecord(r[q]); lastRec = r[q]; // output record r[q]
18.            // input new record into tree
19.            if (end of input) rn[q] = rmax + 1;
                          ...
```

Run Generation using a loser tree : Code (Cont.)

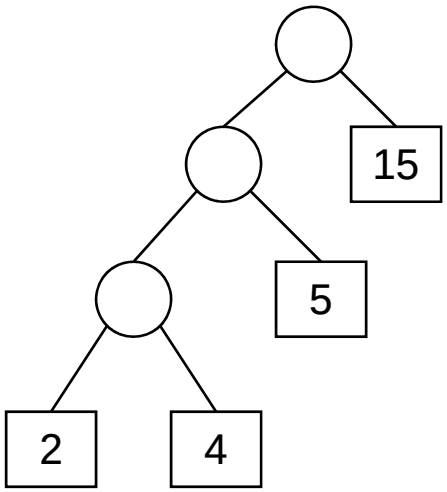
```
                        ...
20.          else {
21.               ReadRecord(r[q]);
22.               if (r[q] < lastRec)  // new record belongs to next run
23.                    rn[q] = rmax = rq + 1;
24.               else rn[q] = rc;
25.          }
26.          rq = rn[q];
27.          // adjust losers
28.          for ( t=(k+q)/2; t; t/=2)  // t is initialized to be parent of q
29.               if ((rn[l[t]] < rq) || ((rn[l[t]] == rq) && (r[l[t]] < r[q])))
30.               { // t is the winner
31.                    swap(q, l[t]);
32.                    rq = rn[q];
33.               }
34.     }
35.     delete [] r; delete [] rn; delete [] l;
36. }
```

Program 7.22: Run generation using a loser tree

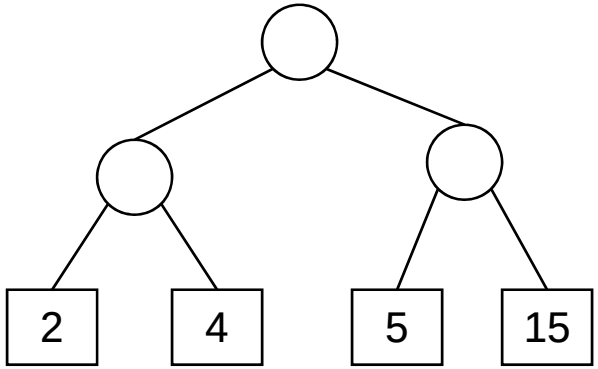
- Analysis of *Runs* : When the input list is already sorted, only one run is generated. On the average, the run size is almost 2k. The time required to generate all the runs for an r run list is O(nlogk), as it takes O(logk) time to adjust the loser tree each time a record is output

7.10.5 Optimal Merging of Runs

- The runs generated by *Runs* may not be of the same size
- When runs are of different size, the merging does not yield minimum runtimes
- The merging runtimes depend on the order of merging



(a)



(b)

Figure 7.27: Possible two-way merges

7.10.5 Optimal Merging of Runs (Cont.)

- The number of merges that an individual records is involved in is
  - Given by the distance of the corresponding external node from the root
  - ex) In Fig 7.27, the records of the run with 15 records are involved in
    - (a) : one merge
    - (b) : two merge
  - ex) In Fig 7.27, total numbers of merges are
    - (a) : 2*3 + 4*3 + 5*2 + 15*1 = 23
    - (b) : 2*2 + 4*2 + 5*2 + 15*2 = 52
- We shall consider the case k = 2 only

Decord tree

- A decord tree is a binary tree in which external nodes represent messages
- The binary bits determine the branching needed at each level

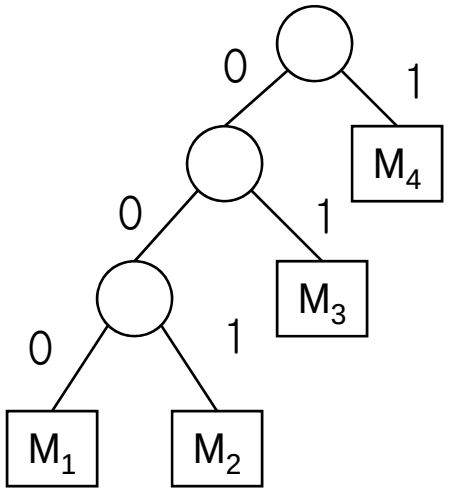


Figure 7.28: A decord tree

- A zero as a left branch and a one as a right branch
- 000, 001, 01, and 1 for message M_1, M_2, M_3, and M_4
  - Huffman codes

Huffman codes

- Expected decoding time for messages M_1, …, M_{n+1}

$$\sum_{1 \le i \le n+1} q_i d_i$$

  - d_i is the distance of the external node for message Mi from the root node.
  - q_i is the relative frequency with which message M_i will be transmitted

- A solution to the problem of finding minimum weighted external path length has been given by D. Huffman
  - begins with a min heap of n single-node tree
  1. extract two minimum-weight trees a and b
  2. combine a and b into a single binary tree c by creating a new root whose left and right subtrees are a and b respectively
  3. weight of c is the sum of the weight of a and b
  4. insert c into the min-heap
  - Repeat step 1 ~ 4 for n-1 times

Huffman

```
template <class T>
void Huffman(MinHeap<TreeeNode <T> *> heap, int n)
{ // heap is initially a min heap of n single-node binary trees as described above
    for (int i=0; i<n-1; i++)
    {  // combine two minimum-weight trees
        TreeNode <T> *first = heap.Pop();
        TreeNode <T> *second = heap.Pop();
        TreeNode <T> *bt = new BinaryTreeNode <T>(first, second,
                                                  first.data+second.data);
        heap.Push((bt);
    }
}
```

Program 7.23: Finding a binary tree with minimum weighted external path length

Example 7.15

q_1=2, q_2=3, q_3=5, q_4=7, q_5=9, q_6=13

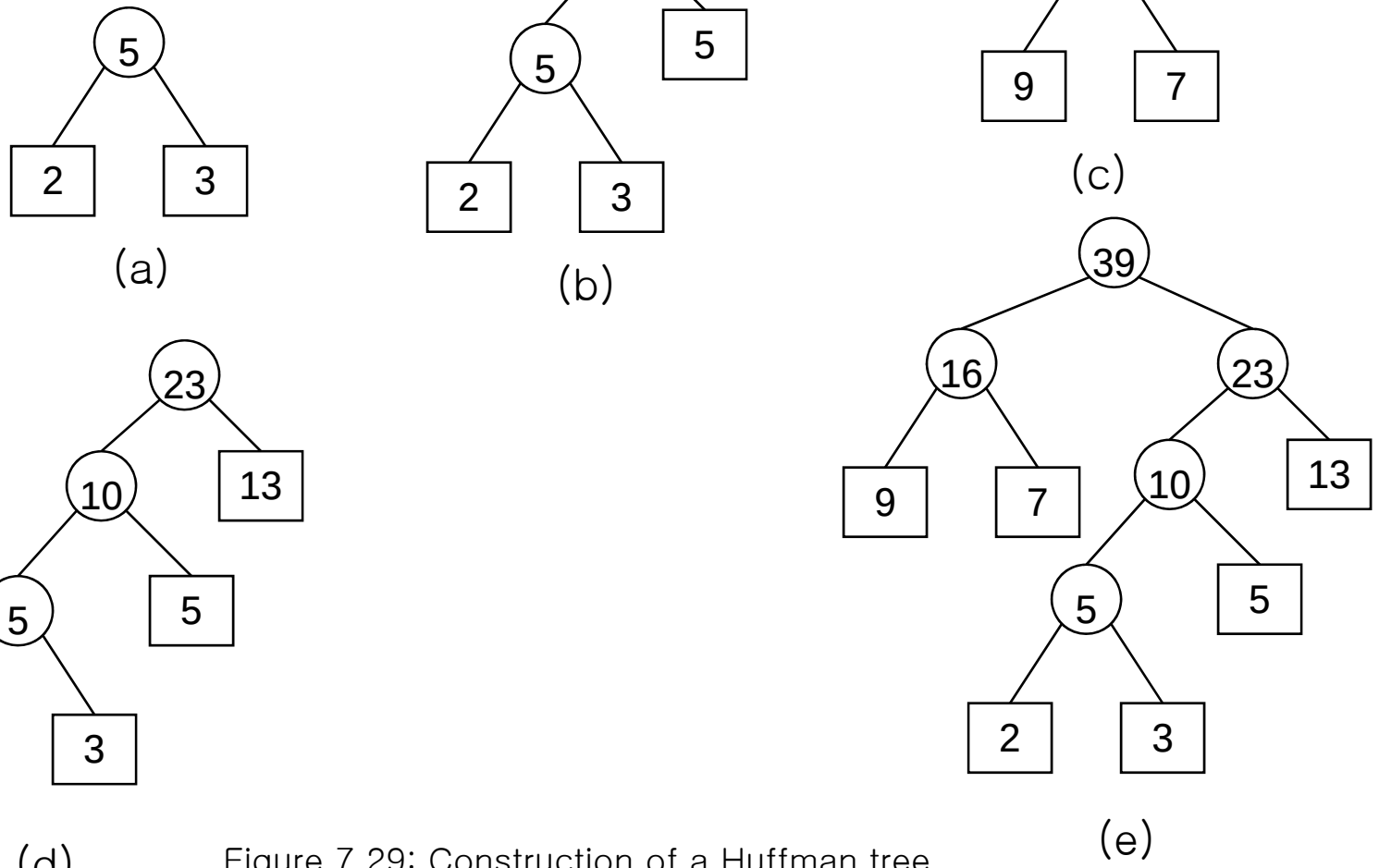
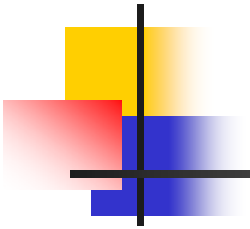


Figure 7,29: Construction of a Huffman tree



Analysis of Huffman

- The main loop is executed n-1 times
- Each call to Pop and Push requires O(logn) time
- Hence, the asymptotic computing time is O(nlogn)