Chapter 7
Sorting

7.1 Motivation

In data structures, the order among data elements is a important relation and sorting becomes the most frequent computing task.

list --- a collection of records, each record having one or more fields.

key --- the fields used to distinguish among records.

Two important uses of sorting:

(1) Binary search a sorted list with n records in O(log n), much better than sequential search an unsorted list in O(n).

(2) Comparing two lists of n and m records containing data that are essentially the same but from different sources. If sorted we can do it in O(n+m), otherwise we need O(nm) time.

Actually there are much more applications of sorting, e.g., query optimization, job scheduling, etc..

Consequently, sorting problem has been extensively studied, and there are many methods proposed.

We shall study several typical methods, indicating when one is superior to others.

Now let us formally state the sorting problem.

- a list of records (R_1, R_2, …, R_n)
- each R_i has key value K_i
- assume an ordering relation (<) on the keys, so that for any 2 key values x and y, x=y or x<y or x>y. < is transitive.

The **sorting problem** is that of finding a permutation, σ, such that $K_{\sigma(i)} \leq K_{\sigma(i+1)}$, $1 \leq i \leq n-1$. The desired ordering is ($R_{\sigma(1)}$, $R_{\sigma(2)}$, …, $R_{\sigma(n)}$).

Let σ_s be the permutation with the following properties:

(1) $K_{\sigma s(i)} \leq K_{\sigma s(i+1)}$, $1 \leq i \leq n-1$.

(2) If $i<j$ and $K_i = K_j$ in the input list, then R_i precedes R_j in the sorted list.

A sorting method that generates the permutation σ_s is **stable**.

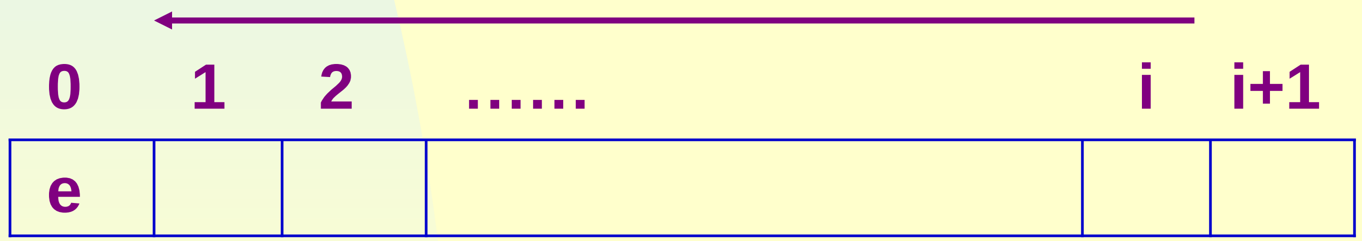
- Sorting
  - internal
    - insert
    - quick
    - merge
    - heap
    - radix
  - external
    - m-way merge
    - run generation

Throughout, we assume that relational operators have been overloaded so that record comparison is done by comparing their keys.

7.2 Insert Sort

Basic step: insert e into a sorted sequence of i records in such a way that the resulting sequence of size i+1 is also ordered.

Uses a[0] to simplify the while loop test.



```
template<class T>
viod 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 for at least i+2 elements.
     a[0]=e;
     while (e < a[i])   //stable
     {
         a[i+1]=a[i];
         i--;    // a[i+1] is always ready for storing element
      }
      a[i+1]=e;
}
```

Insertion sort:
Begin with the ordered sequence a[1], then successively insert a[2], a[3], …, a[n] into the sequence.

```
template<class T>
void InsertionSort (Element *a, const int n)
{ //sort a[1:n] into nondescreasing order.
    for (int j=2; j<=n; j++) {
        T temp = a[j];  // necessary, because a[j] may change in
                        // Insertion
        Insert (temp, a, j-1);
    }
}
```

Analysis of insert sort:

(1) The worst case

Insert(e, a, i) makes i+1 comparisons before making insertion --- O(i).

InsertSort invokes Insert for i=j-1=1, 2,…,n-1, so the overall time is

$$O\left(\sum_{i=1}^{n-1}(i+1)\right)=O(n^2).$$

(2) Estimate of the actual computing time

R_i is **left out of order** (LOO) iff $R_i < \max_{1 \le j < i} \{ R_j \}$.

For every record, at least O(1) is needed. If k is the number of LOO records, the computing time is O(n+kn).

It can also be shown that the average time is $O(n^2)$.

When k<<n, this method is very desirable. And for $n \le 30$, it is the fastest.

Example 7.1: n=5, key sequence is 5, 4, 3, 2, 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**

Example 7.2: **n=5, key sequence is 2, 3, 4, 5, 1**

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

InsertSort is stable.

Variations:

1 Binary Insert Sort.

2 Linked Insert Sort.

Exercises: P401-1, 3

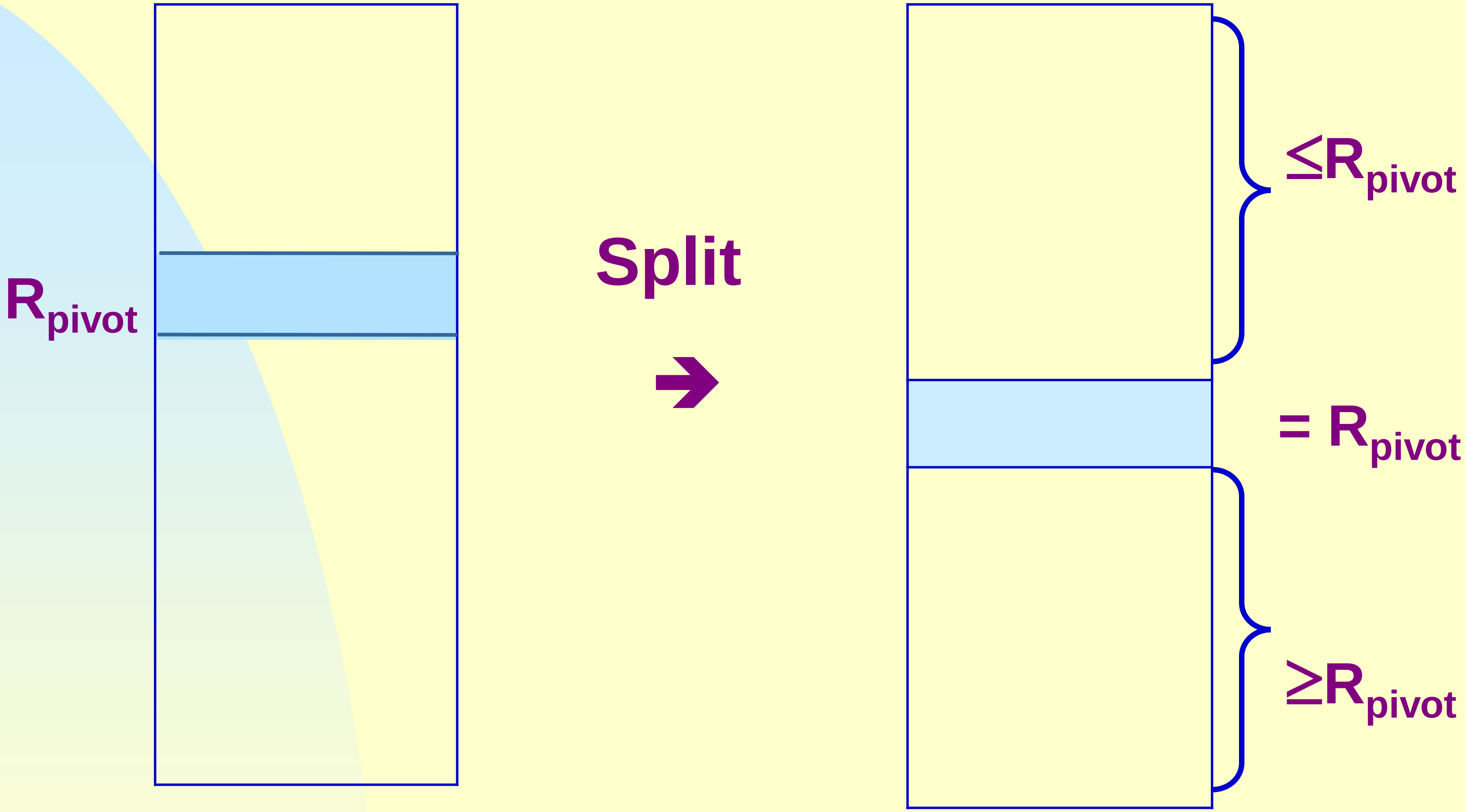
7.3 Quick Sort

The quick-sort has the best average behavior among the sorting methods we shall be studying.

Idea: given a list (R_{left}, R_{left+1}, …, R_{right}), select a pivot record from among R_i (left $\le$ i $\le$ right) , put the pivot in the correct spot s(pivot) such that after the positioning,

$$R_j \le R_{s(pivot)} \quad \text{for } j<s(pivot)$$

$$R_j \ge R_{s(pivot)} \quad \text{for } j>s(pivot)$$



R_{pivot}
Split
$\leq R_{pivot}$
$= R_{pivot}$
$\geq R_{pivot}$

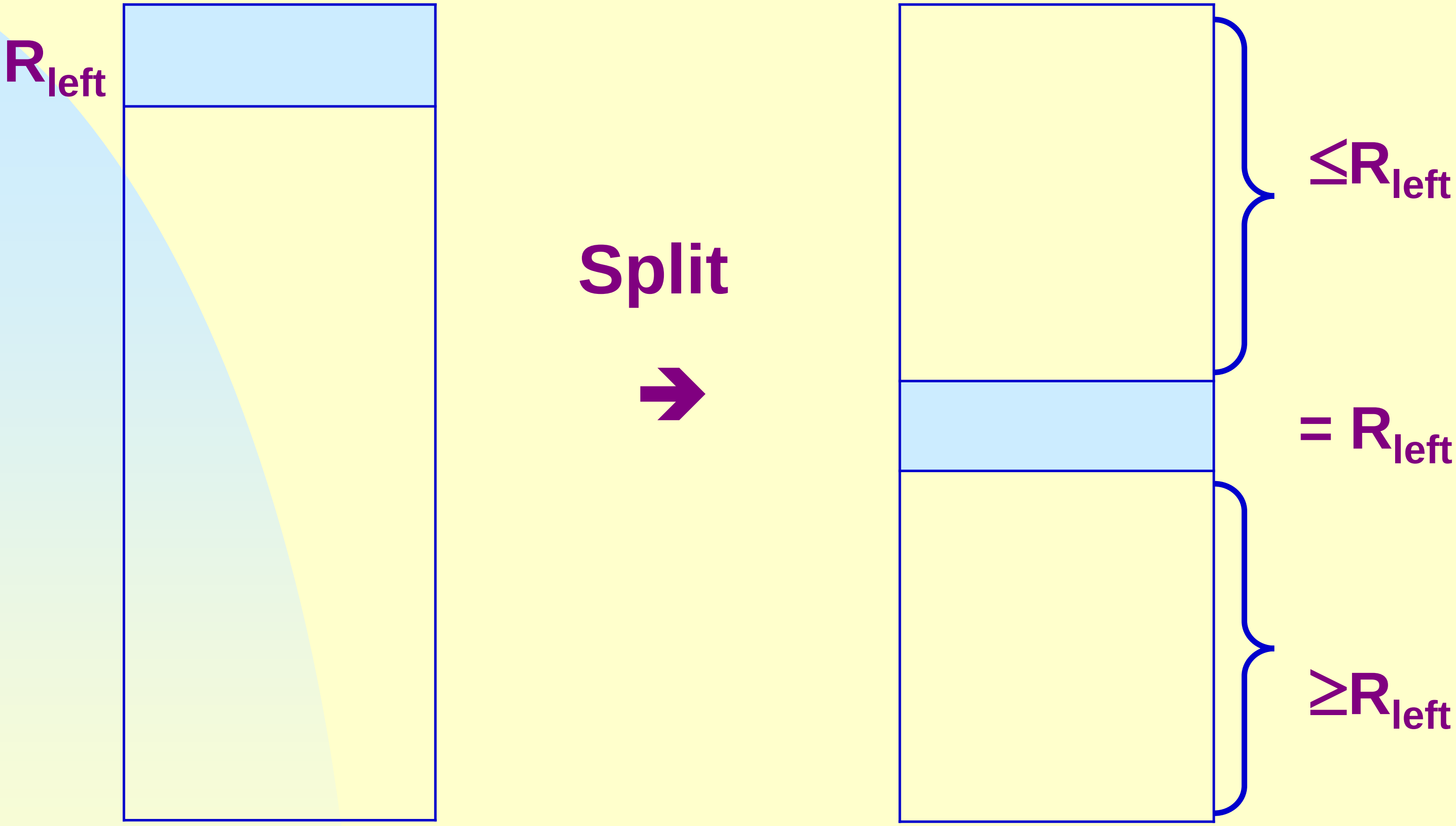
The original list is partitioned into 2 sublists:

(R_{left}, …, $R_{s(pivot)-1}$) and

($R_{s(pivot)+1}$, …, R_{right})

and they may be sorted independently.

For simplicity, we just choose R[left] as the pivot.


R_{left}
Split
$\leq R_{left}$
$= R_{left}$
$\geq R_{left}$

```
template <class T>
void QuickSort (T *a, const int left, const int right)
{ // Sort a[left:right] into non-decreasing order. a[left] is
  // arbitrarily chosen as the pivot. Assume 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);
    }
}
```

To sort a[1:n], invoke QuickSort(a, 1, n).

Example 7.3:

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		

Analysis of QuichSort:

(1) The worst case

2 sublists with size n-1 and 0.

$$T_{worst}(n) \le cn + T_{worst}(n-1)$$

$$\le cn + c(n-1)+T_{worst}(n-2)$$

$$\dots$$

$$\le c \sum_{i=1}^{n} i + T_{worst}(0)=cn(n+1)/2+d= O(n^2).$$

(2) The optimal case

2 sublists with equal size, roughly n/2.

$$T_{opt}(n) \le cn + 2T_{opt}(n/2)$$

$$\le cn + 2(cn/2+2T_{opt}(n/4))$$

$$\le 2cn + 4T_{opt}(n/4)$$

$$\dots$$

$$\le cn\log_2 n+nT_{opt}(1) = O(n\log n).$$

(3) The average time

Lemma 7.1: $T_{avg}(n) \le k*n\log_e n$ for $n \ge 2$.

Proof:

In the call to QuickSort(a, 1, n), the pivot gets place at position j, we need to sort 2 sublists of size j-1 and n-j. j may take on any of the 1 to n with equal probability, so

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

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

Let $T_{avg}(0) \le b$ and $T_{avg}(1) \le b$ for some constant b,

k=2(b+c),

we show $T_{avg}(n) \le knlog_e n$ for $n \ge 2$.

Induction on n.

n=2, from (7.1) $T_{avg}(2) \le 2c+2b \le k*2log_e 2$.

Assume $T_{avg}(n) \le k*nlog_e n$ for $2 \le n < m$.

Then

$$T_{avg}(m) \le cm + \frac{2}{m}\sum_{j=0}^{m-1} T_{avg}(j)$$

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

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

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

$$T_{avg}(m) \le cm + \frac{4b}{m} + \frac{2k}{m}\int_2^m x\log_e x\,dx$$

$$= cm + \frac{4b}{m} + \frac{2k}{m}\left[\frac{m^2\log_e m}{2} - \frac{m^2}{4}\right]$$

$$= cm + \frac{4b}{m} + km\log_e m - \frac{km}{2}$$

$$= km\log_e m + \left[cm + \frac{4b}{m} - \frac{km}{2}\right]$$

For m≥2

$$cm+\frac{4b}{m}-\frac{km}{2}=cm+\frac{4b}{m}-\frac{2(b+c)m}{2}$$

$$=\frac{4b}{m}-bm\leq 0$$

Therefore For m≥2, $T_{avg}(m) \leq kmlog_e m$.

Stack space requirement:

- **best case: split evenly --- O(log n)**
- **worst case: split into a left sublist of size n-1 and a right of 0 --- O(n)**
- **if we use only one recursion and other parts are replaced by iteration, and sort the smaller sublist first, let S(n) be the space for list of size n, then**
    - **$S(1)=c$ (c is a constant)**
    - **$S(n) \le c+S(n/2)=c \log n+S(1)=O(\log n)$**

- **better choice of the pivot:**

  pivot=median $\{R_{left}, K_{(left+right)/2}, R_{right}\}$

Note also that quick sort is unstable.

Exercises: P405-1, 2, 5

7.5 Merge Sort

7.5.1 Merging

Merge 2 sorted lists to get a single sorted list.

i1

$(X_l, \ \ldots, \ \ X_m)$

iResult

$\rightarrow \ (Z_l, \ \ldots, Z_n)$

$(X_{m+1}, \ldots, X_n)$

i2

```
template <class T>
void Merge (T *initList, T *mergedList,
                                        const int l, const int m, const int n)
{ // two sorted lists initList[l:m] and initList[m+1:n] are
  // merged to obtain the sorted list mergedList[l:n].
    for (int i1=l, iResult=l, i2=m+1; i1<=m && i2<=n; iResult++)
       if (initList[i1] <= initList[i2])
           mergedList[iResult]=initList[i1++];  //stable
       else
           mergedList[iResult]=initList[i2++];
    // copy remaining records , if any, of the first list
    copy(initList+i1, initList+m+1, mergedList+iResult);
    // copy remaining records , if any, of the second list
    copy(initList+i2, initList+n+1, mergedList+iResult);
}
```

Analysis of Merge:

At each iteration of the for loop, either i1++ or i2++, the for loop is iterated at most n-l+1 times.

At most n-l+1 records are copied.

The total time: O(n-l+1).

If each record has a size s, then the total time is O(s(n-l+1)).

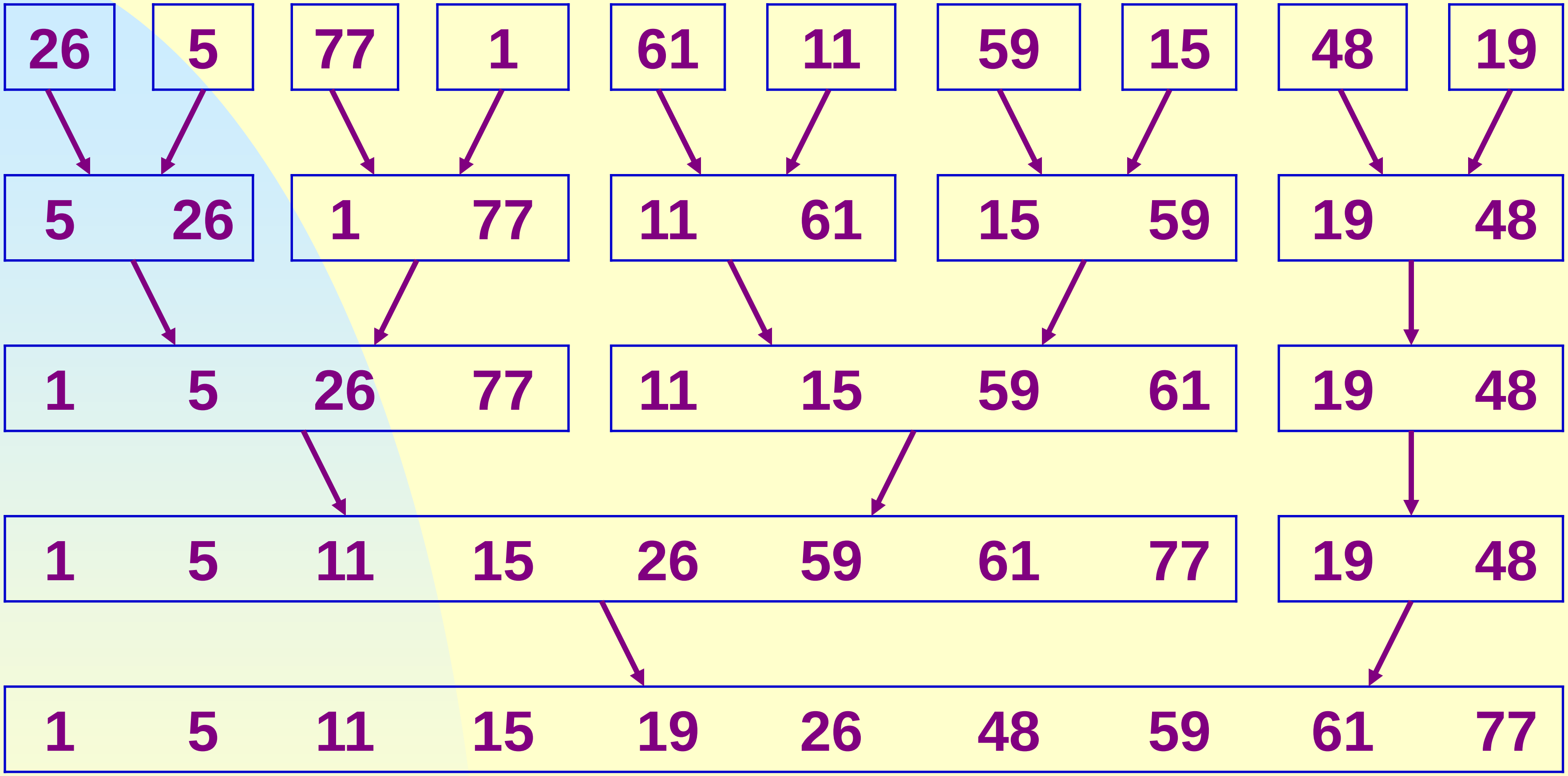
When s>1, use linked lists, only need n-l+1 links, the merge time becomes O(n-l+1) and is independent of s.

7.5.2 Iterative Merge Sort

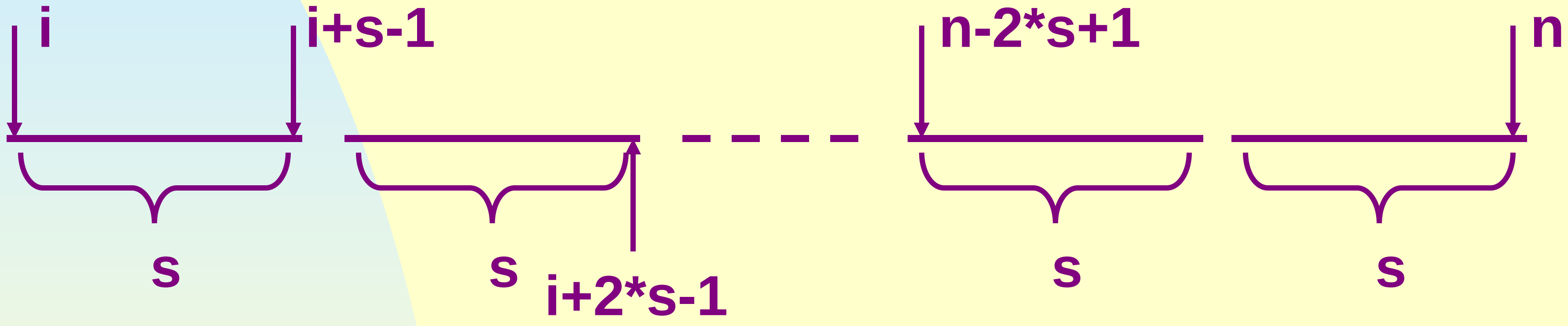
Basic idea:

At the beginning, interpret the input as n sorted sublists, each of size 1. These lists are merged by pairs to obtain n/2 lists, each of length 2 (if n is odd, then one list is of length 1). These n/2 lists are then merged by pairs, and so on until only one list is get.

As shown in the following:


26 5 77 1 61 11 59 15 48 19
5 26 1 77 11 61 15 59 19 48
1 5 26 77 11 15 59 61 19 48
1 5 11 15 26 59 61 77 19 48
1 5 11 15 19 26 48 59 61 77

A merge sort consists of several passes, it is convenient to write a function MergePass for this.



```
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.
    for (int i=1; i<=n-2*s+1; //are there 2*s elements?
                              i+=2*s)
        Merge(initList, resultList, i, i+s-1, i+2*s-1);

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

Now the sort can be done by repeatedly invoking MergePass.

```
template <class T>
void MergeSort (T *a, const int n)
{ // Sort a[1:n] into nondereasing 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);// last pass may just copy
    }
    delete [ ] tempList;
}
```

Analysis of MergeSort:

Makes several passes. After the 1st pass, the result sublists are of size $2=2^1$. After the ith pass, the size is 2^i. Consequently, a total of $\lceil \log_2 n \rceil$ passes are made, each takes o(n), the total time is O(n log n).

It is easy to verify that MergeSort is stable.

Exercises: P412-1

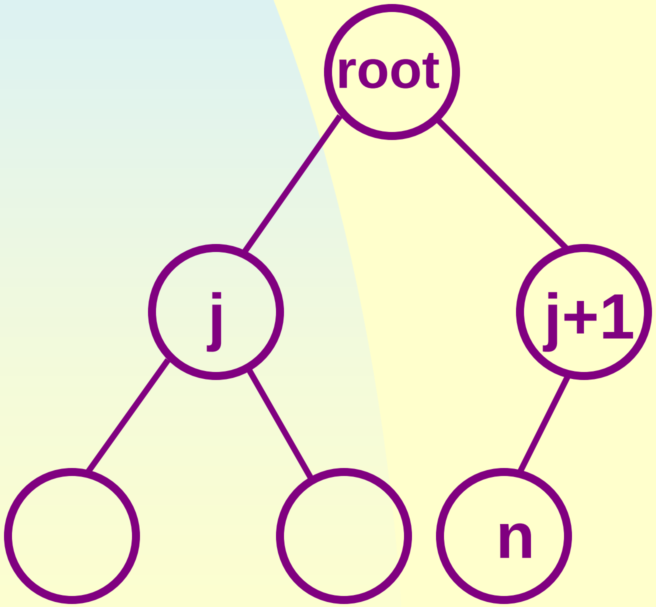
7.6 Heap Sort

Merge sort has a computing time of O(n log n), both in worst and average case, but it requires O(n) additional storage.

Heap sort requires o(1) additional space, and also has as its worst and average computing time O(n log n).

In heap sort, we utilize the max-heap structure in chapter 5 .

To create and recreate a heap efficiently, we need a function **Adjust**, which starts with a binary tree whose left and right subtrees are max heaps and adjusts the entire binary tree into a max heap.



j/2 is always ready for storing record

```
template <class T>
void adjust (T *a, const int root, const int n)
{ // 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 the lager child of its parent
      if (e>=a[j]) break;
      a[j/2]=a[j];  //move jth record up the tree
    }
    a[j/2]=e;
}
```

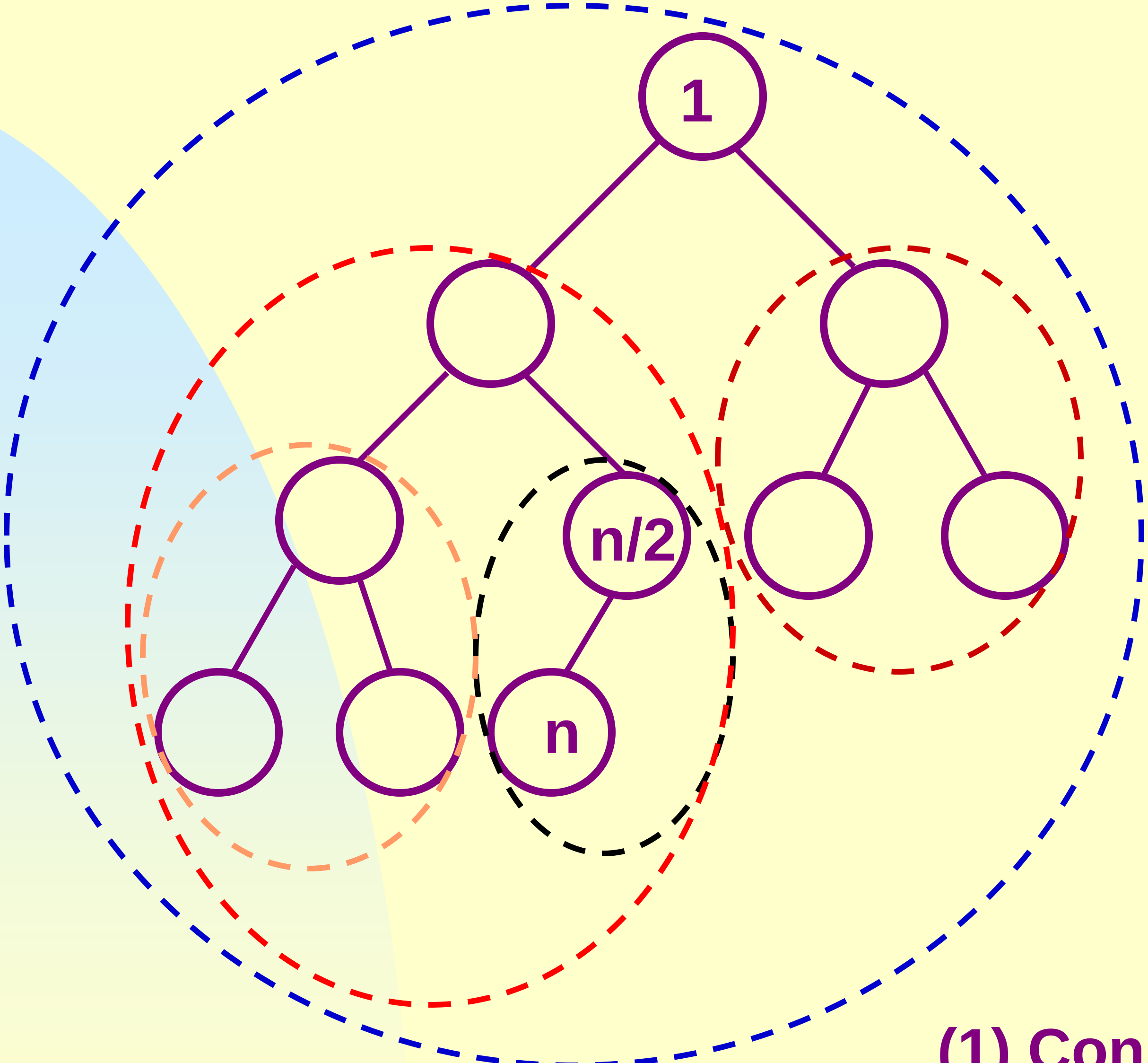
Analysis of adjust:

If the depth of tree is d, the for loop is executed at most d times. Hence, the computing time is O(d).

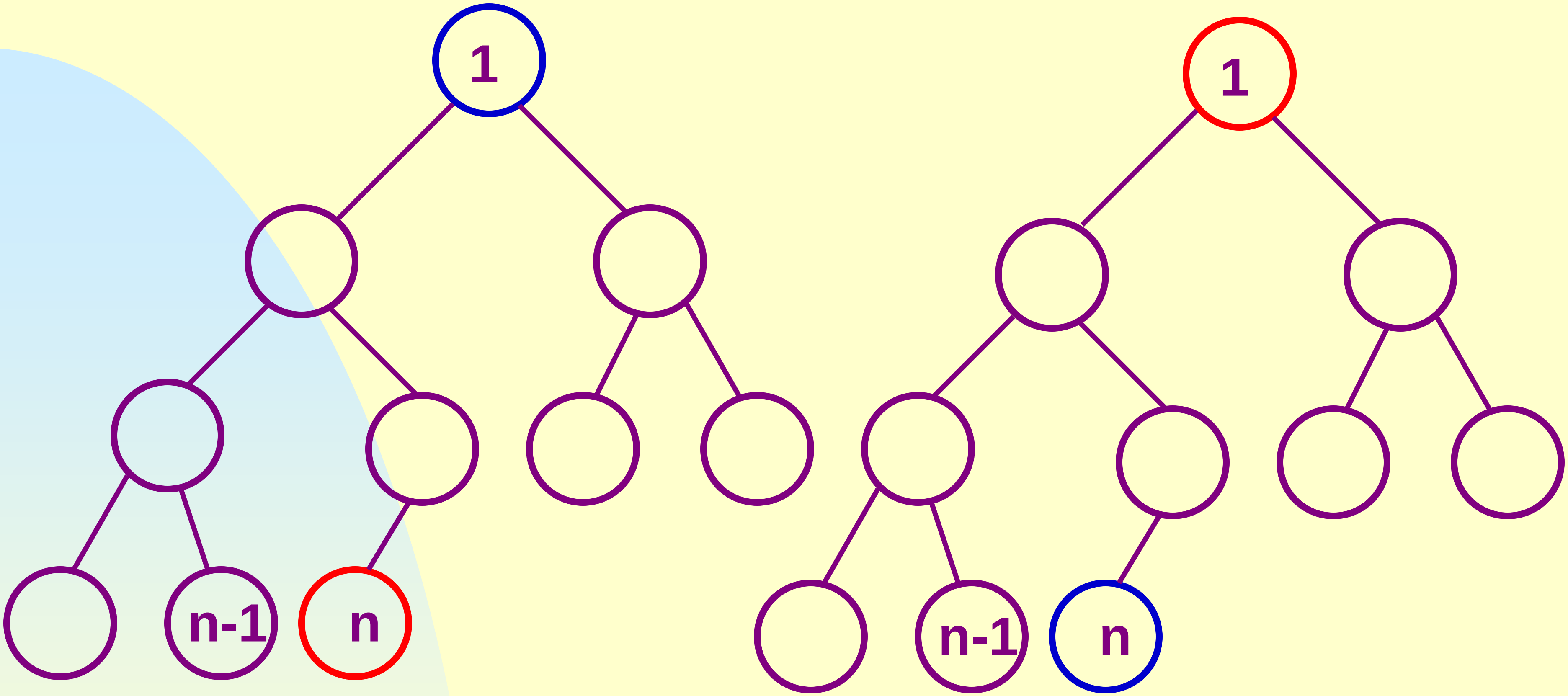
To sort the list

(1) Create a max heap by using Adjust.

(2) Make n-1 passes, in each pass, swap the first and last records in the heap, and decrement the heap size and readjust it.



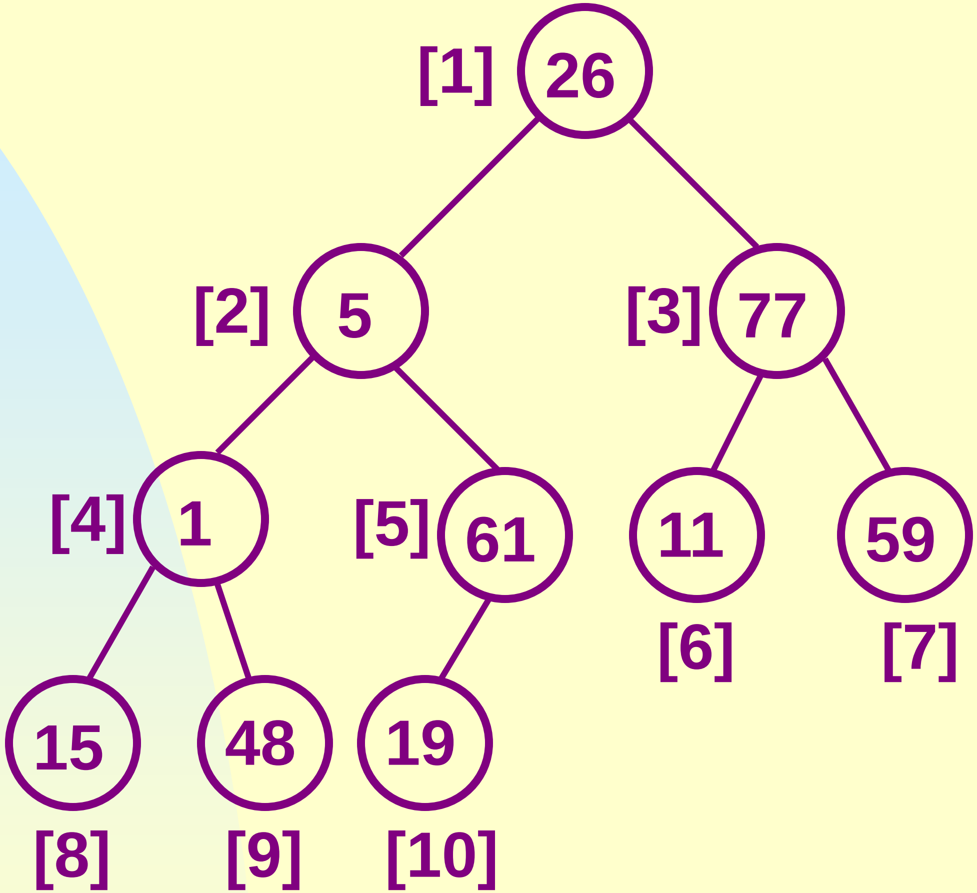
(1) Construct a heap



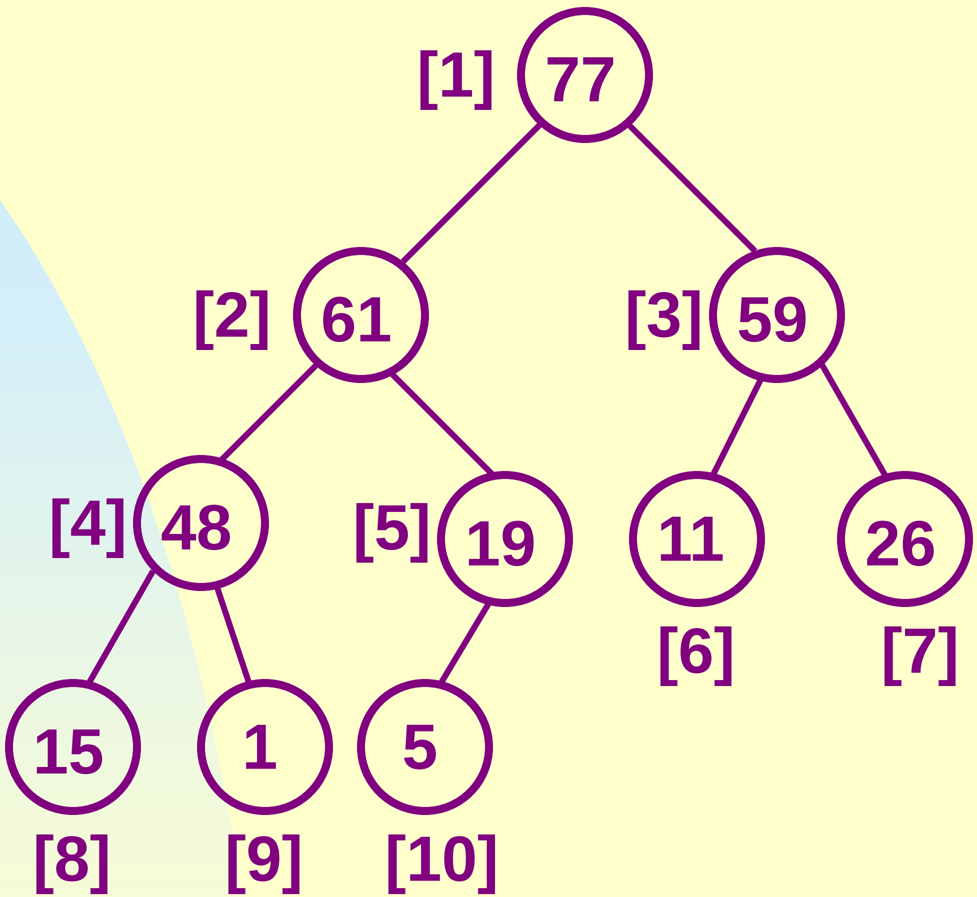
(2) Swap and readjust

```
template <class T>
void HeapSort (T *a, const int n)
{ // Sort a[1:n] into nondeceasing order.
    for (int i=n/2; i>=1; i--)   // convert list into a heap
        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);         // recreate heap
    }
}
```

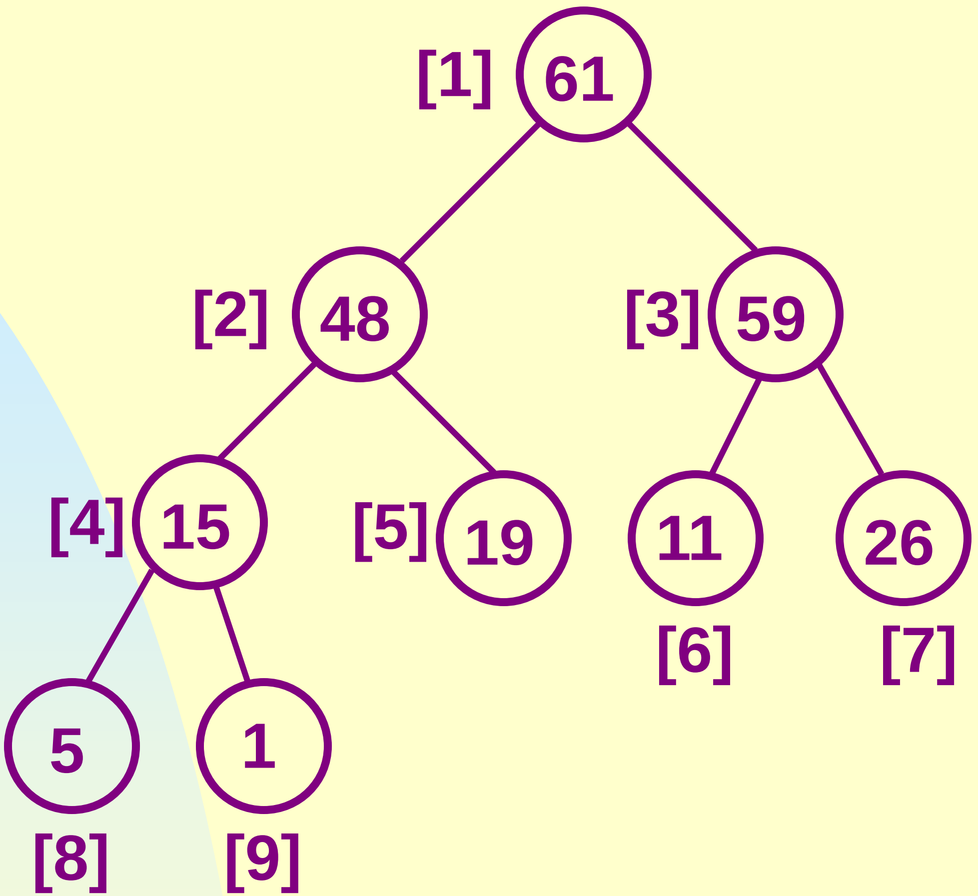
Example 7.7:



(a) Input list

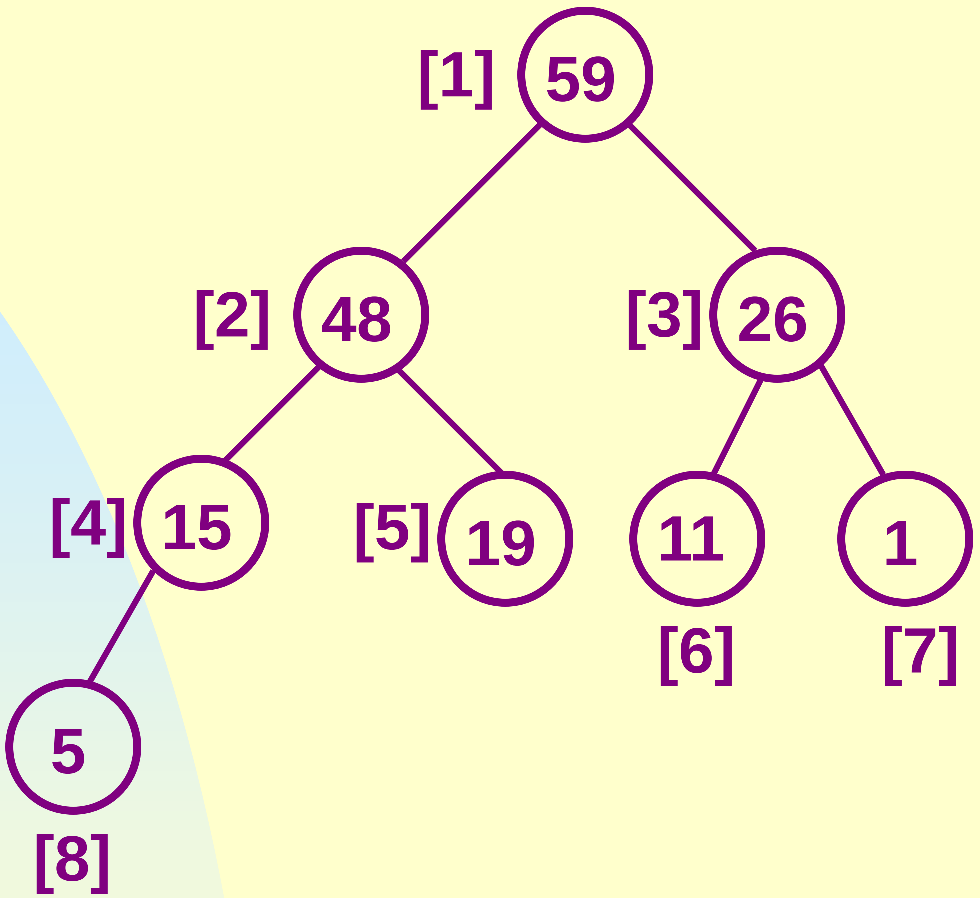


(b) Initial heap



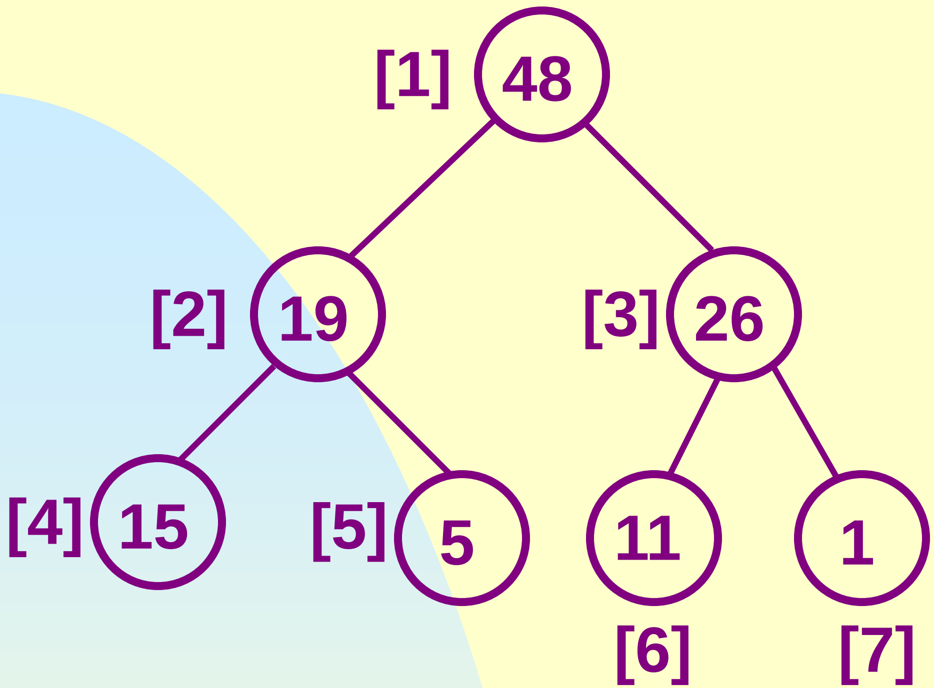
(c) Heap size=9

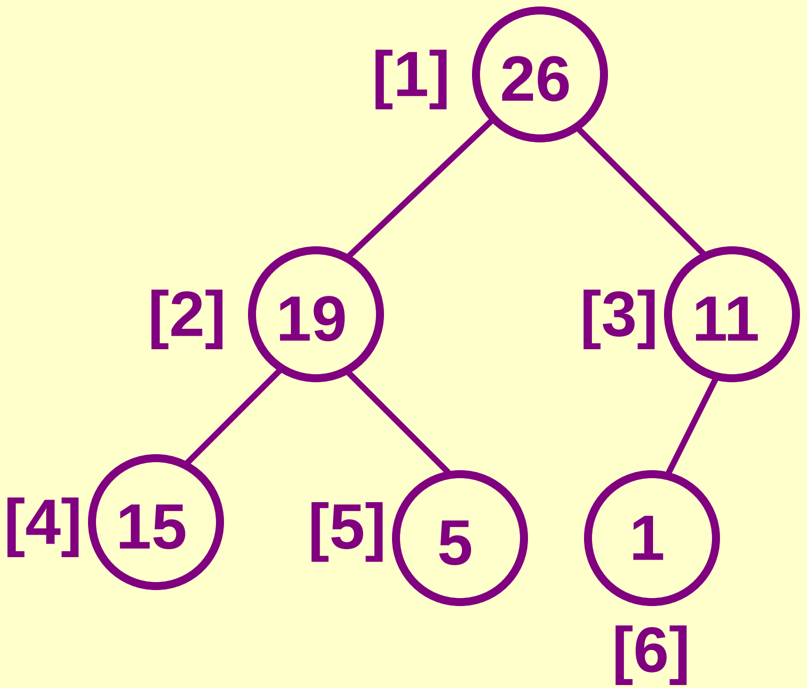
Sorted=[77]



(d) Heap size=8

Sorted=[61, 77]



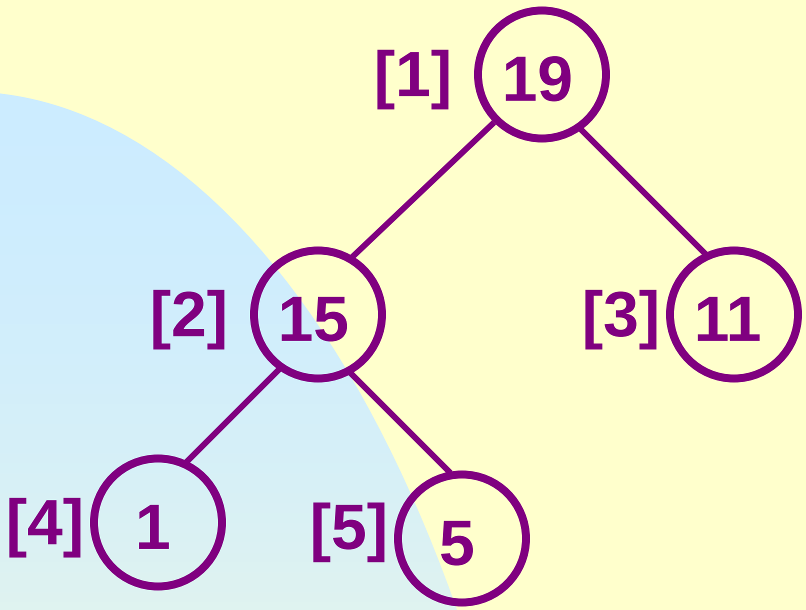


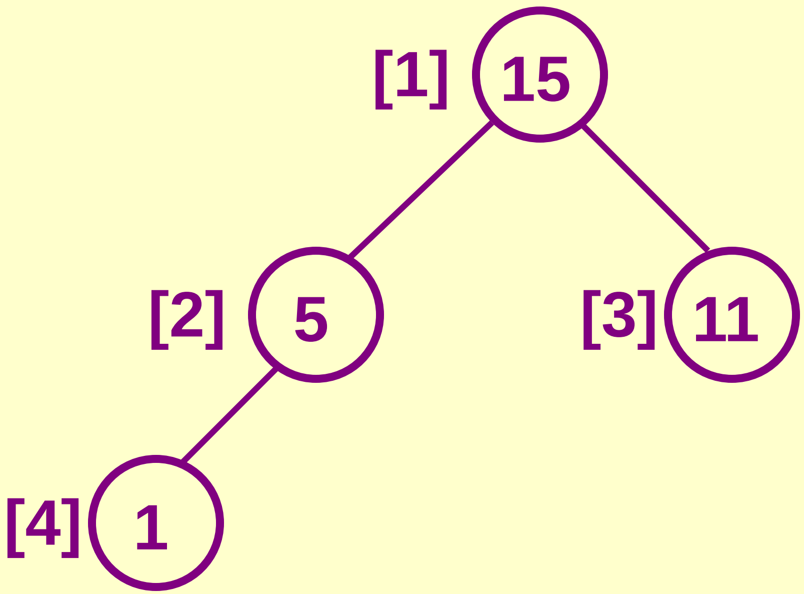
(e) Heap size=7

Sorted=[59, 61, 77]

(f) Heap size=6

Sorted=[48, 59, 61, 77]



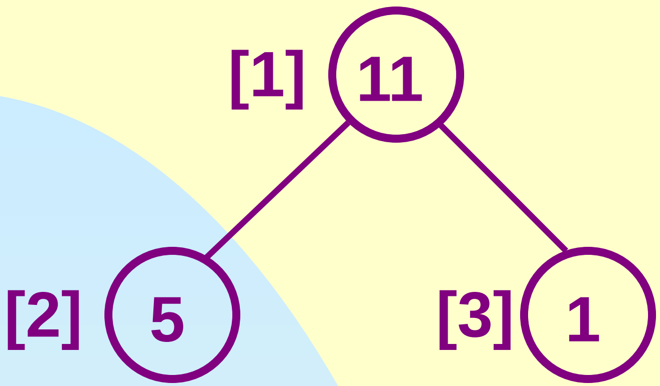


(g) Heap size=5

[26, 48, 59, 61, 77]

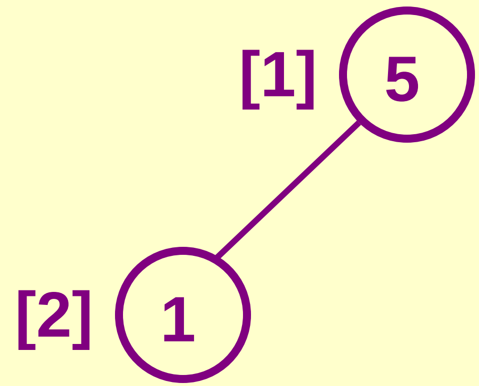
(h) Heap size=4

[19, 26, 48, 59, 61, 77]



(i) Heap size=3

[15, 19, 26, 48, 59, 61, 77]



(j) Heap size=2

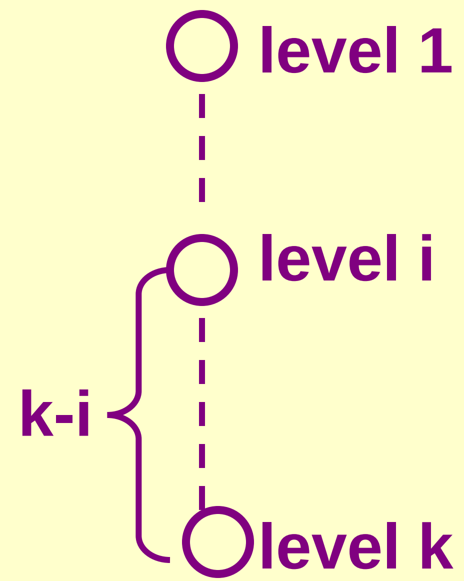
[11, 15, 19, 26, 48, 59, 61, 77]

(j) Heap size=1

[5, 11, 15, 19, 26, 48, 59, 61, 77]

Analysis of HeapSort:

- suppose $2^{k-1} \le n < 2^k$, the tree has k levels.
- the number of nodes on level i $\le 2^{i-1}$.
- in the first loop, Adjust is called once for each node that has a child, hence the time is no more than



$$\sum_{1\le i\le k-1} 2^{i-1}(k-i) =$$

$$\sum_{1\le i\le k-1} 2^{k-i-1} i \le n \sum_{1\le i\le k-1} i/2^i < 2n = O(n)$$

- in the next loop, n-1 applications of Adjust are made with maximum depth $k = \lceil \log_2(n+1) \rceil$.

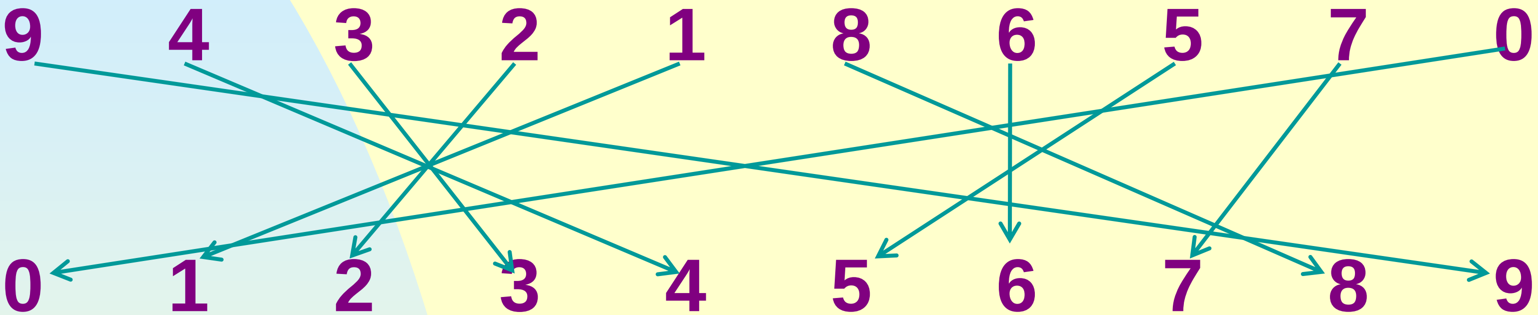
The total time: O(n log n).

Additional space: O(1).

Exercises: P416-1, 2

7.7 Sorting on Several Keys

Observe :



This is the basic idea of bin sort, to make effective use of it, we can interpret a key as several sub-keys.

Problem : to sort records on keys $K^1, K^2, \ldots, K^d$ (K^1 is the most significant key and K^d the least).

A list of records $R_1, R_2, \ldots, R_n$ is sorted with respect to the keys $K^1, K^2, \ldots, K^d$ iff

For every pair of i and j, $i<j$ and

$$(K_i^1, \ldots, K_i^d) \le (K_j^1, \ldots, K_j^d)$$

For example: sort a deck of cards may be regarded as a sort on 2 keys:

K^1 [Suits]:

♣ < ♦ < ♥ < ♠

K^2 [Face values]:

2<3<4<5<6<7<8<9<10<J<Q<K<A

A sorted deck of cards has the following ordering:

2♣,…, A♣,…, 2♠,…, A♠

LSD (least significant digit first) sort :

- sort the cards first into 13 piles corresponding to their face values. Place K’s on top of A’s, … , 2’s on top of 3’s.
- then sort on the suit using a **stable** method to obtain 4 piles, combine the piles to obtain the sorted cards.

To sort on each key K^i, use bin sort, i. e., if $0 \leq K^i < r$, r bins are set up, one for each value of K^i, and records are placed into their corresponding bins. For n records, the time is $O(n+r)$

For one logical key, we can interpret it as being composed of several keys, e.g., if $0 \leq K \leq 999$, K can be considered as $(K^1 K^2 K^3)$, $0 \leq K^i < 10$, $i=1,2,3$, this is also called Radix-10 sort.

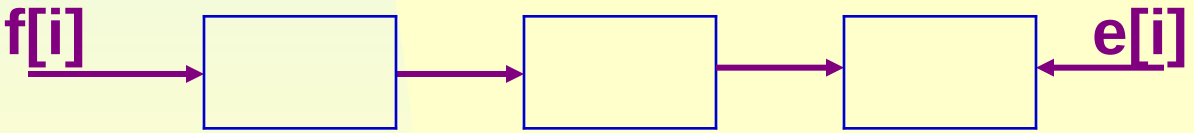
In general, for a Radix-r sort, r bins are needed.

Assume:

(1) records R_1, R_2, …, R_n and their keys are decomposed using a radix of r, each key have d digits in the range of [0, r-1].

(2) records have a link field.

(3) for each bin i, $0 \leq i < r$



```
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(e, i, r) returns
  // the ith radix-r digit (from the left, the first is the 0th digit)
  // of e's key. Each digit is in the range of [0, r).
    int e[r],  f[r];  // queue end and front pointers

    // create initial chain of records starting at first
    if (n==0) return 0;  int first=1;
    for (int i=1; i<n; i++) link[i]=i+1; // linked 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
```

```
            int k=digit(a[current], i, r);
            if (f[k]==0) f[k]=current;
            else link[e[k]]=current;
            e[k]=current;
        }
        for (int j=0; !f[j] ; j++); // find first nonempty queue
        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;
}
```

Analysis of RadixSort:

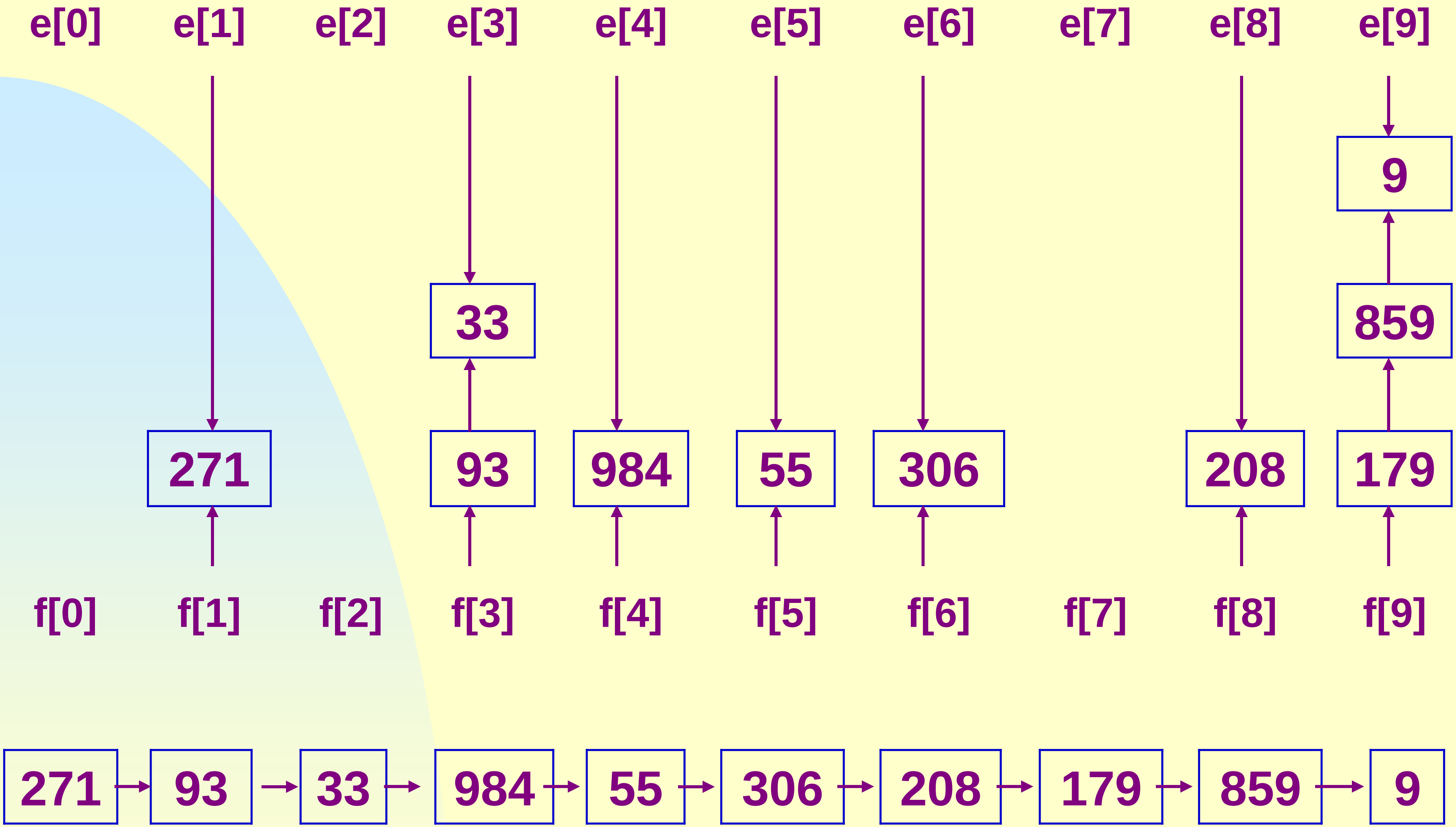
d passes over the data, each taking O(n+r). Hence the total time is O(d(n+r)).

d will depend on r and the largest key.

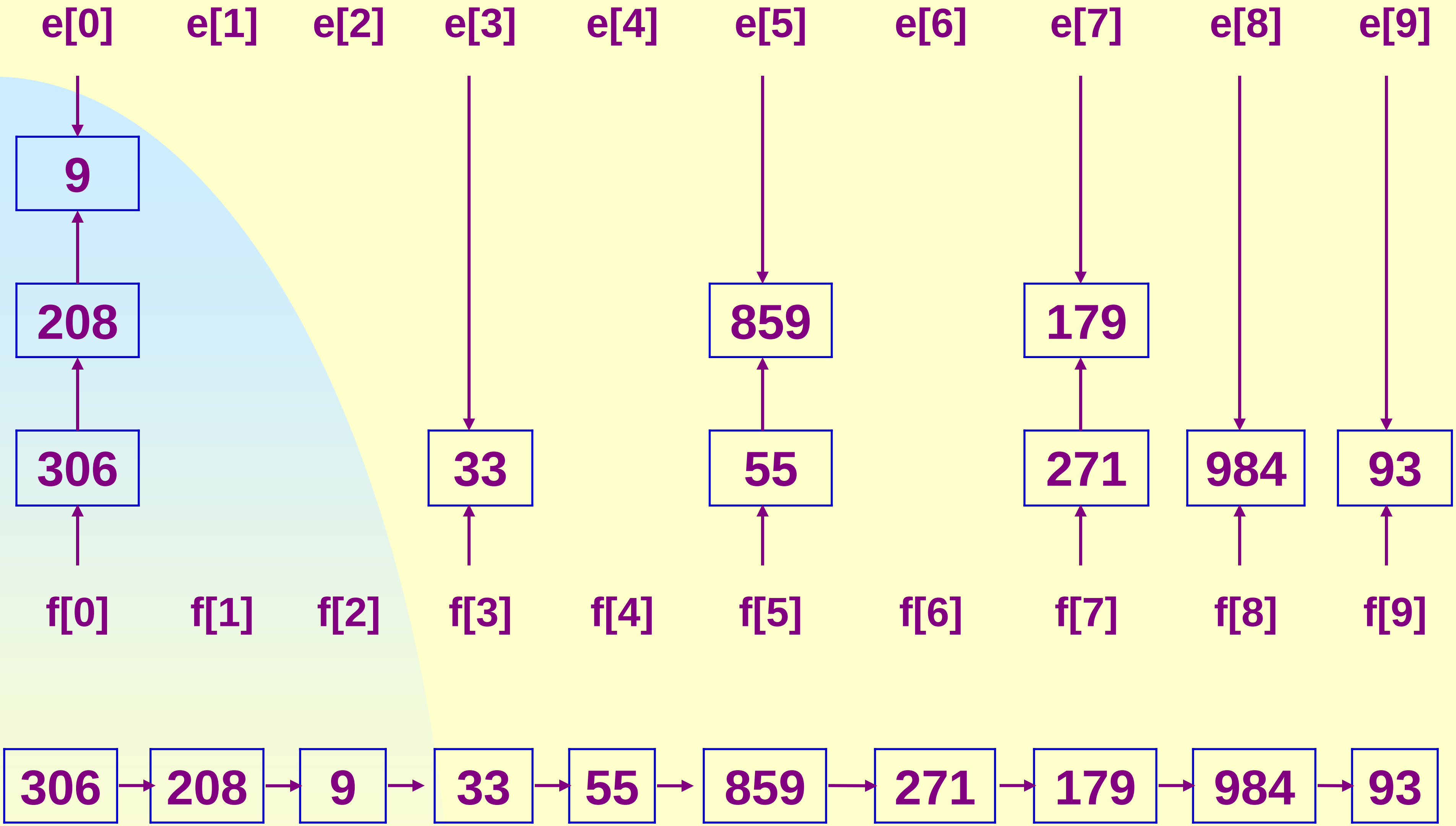
Example 7.8: sort 10 numbers in the range [0,999], r=10, hence d=3.

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
179 →	208 →	306 →	93 →	859 →	984 →	55 →	9 →	271 →	33

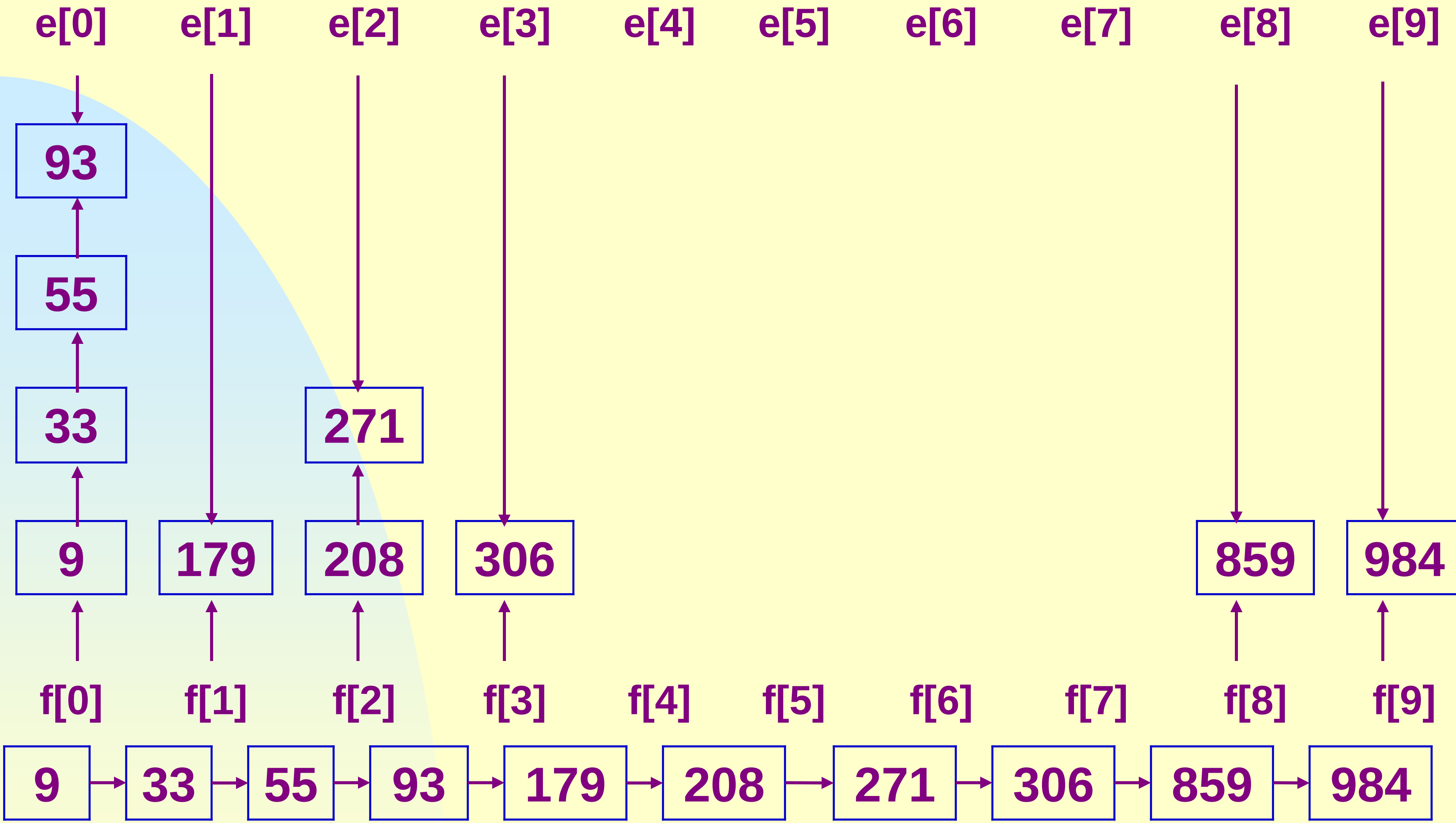
(a) Initial input


e[0] e[1] e[2] e[3] e[4] e[5] e[6] e[7] e[8] e[9]
9
33
859
271
93
984
55
306
208
179
f[0] f[1] f[2] f[3] f[4] f[5] f[6] f[7] f[8] f[9]
271 93 33 984 55 306 208 179 859 9

(b) First pass



(c) second pass


e[0]
e[1]
e[2]
e[3]
e[4]
e[5]
e[6]
e[7]
e[8]
e[9]
93
55
33
271
9
179
208
306
859
984
f[0]
f[1]
f[2]
f[3]
f[4]
f[5]
f[6]
f[7]
f[8]
f[9]
9
33
55
93
179
208
271
306
859
984

(d) third pass

Exercises: P422-1, 3, 5

7.9 Summary of Internal Sorting

The behavior of Radix Sort depends on the size of keys and the choice of r.

The following summarizes the other 4 methods we have studied:

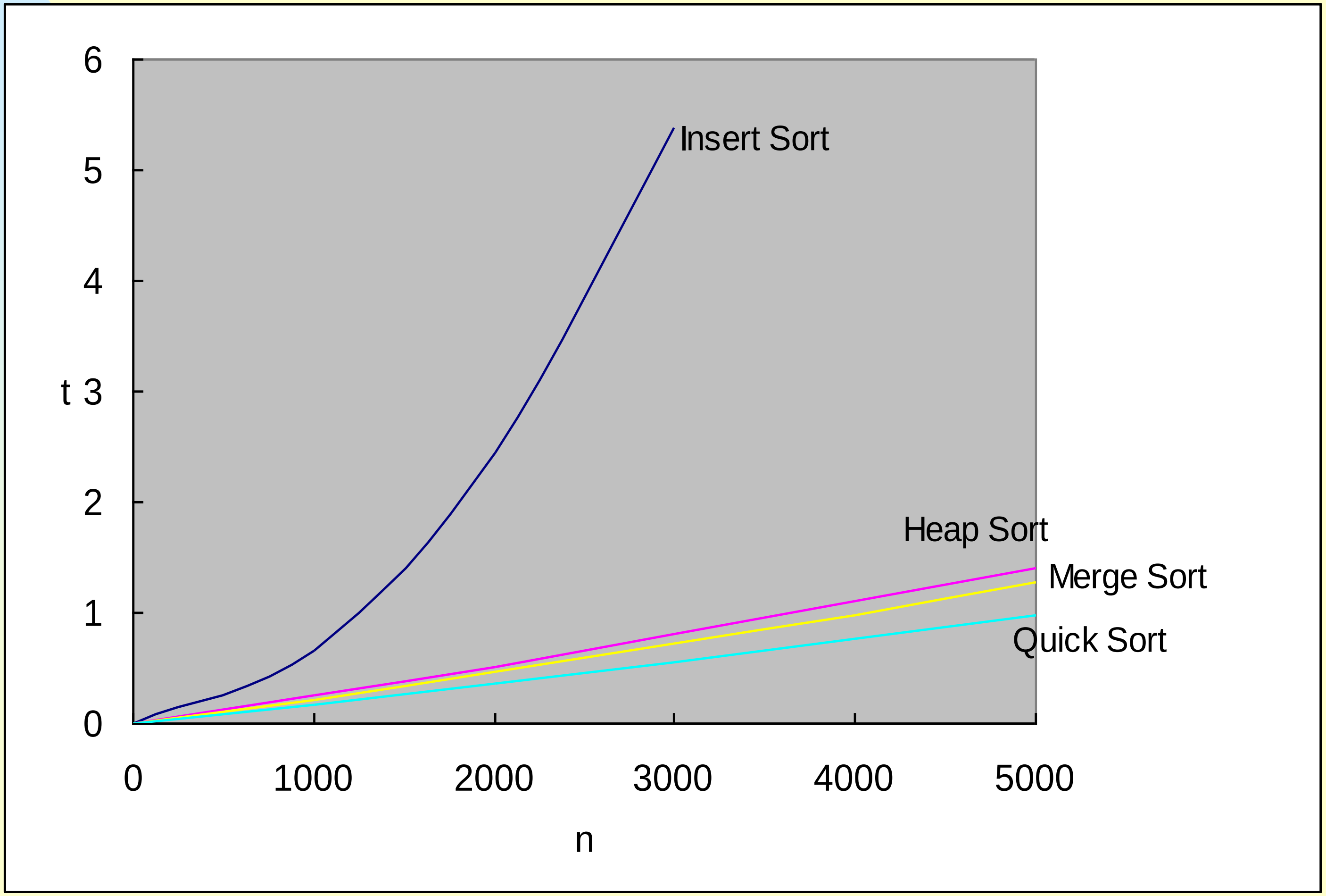
Method	Worst	Average	Working Storage
Insertion Sort	n^2	n^2	O(1)
Heap Sort	n log n	n log n	O(1)
Merge Sort	n log n	n log n	O(n)
Quick Sort	n^2	n log n	O(n) or O(log n)

The next slide gives the average runtimes for the 4 methods on a 1.7G Intel Pentium 4 PC with 512M RAM and Microsoft Visual.NET2003.

For each n at lest 100 randomly generated integer instances were run.

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	0.057	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

The following is a plot of average times (milliseconds) for sort methods:



For average behavior, we can see:

- **Quick Sort outperforms the other sort methods for suitably large n.**
- **the break-even point between Insertion and Quick Sort is near 100, let it be nBreak.**
- **when n < nBreak, Insert Sort is the best, and when n > nBreak, Quick Sort is the best.**
- **improve Quick Sort by sorting sublists of less than nBreak records using Insertion Sort.**

Experiments: P435-4

7.10 External Sorting

7.10.1 introduction

- the lists are too large to be entirely contained in the internal memory, making an internal sort impossible.

- the lists to be sorted resides on a disk.

- block: the unit of data that is read from or written to a disk at one time.

- disk --- provide direct access to block, its I/O time is determined by:

(1) seek time

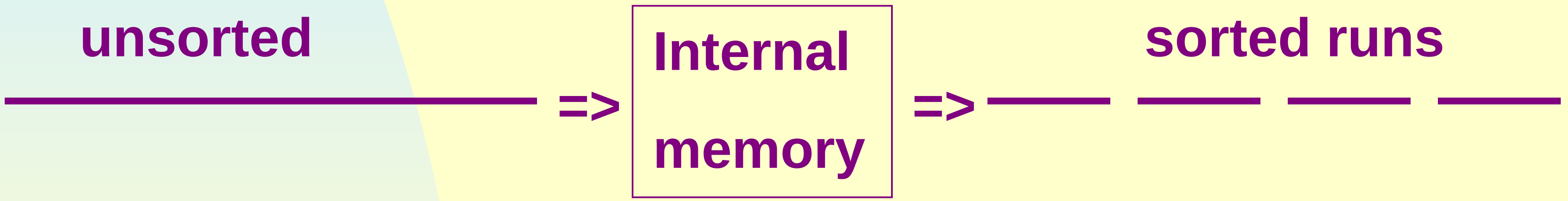
(2) latency time

(3) transmission time

Note I/O rate << CPU rate.

The most popular method for external sorting is merge sort, which consists of 2 phases:

(1) Segments of the input list are sorted in internal memory, these sorted segments, known as runs, are written onto disk as they are generated.



(2) The runs are merged together until only one run left.

Because **Merge** requires only the leading records of runs being merged to be present in memory at one time, it is possible to merge large runs together.

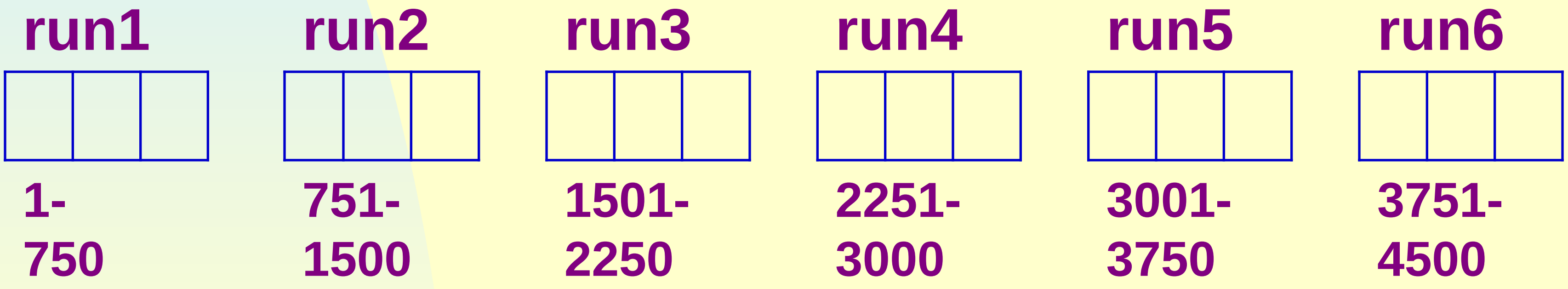
Example 7.12:

- a list of 4500 records to be sorted
- the internal memory is capable of sorting at most 750 records

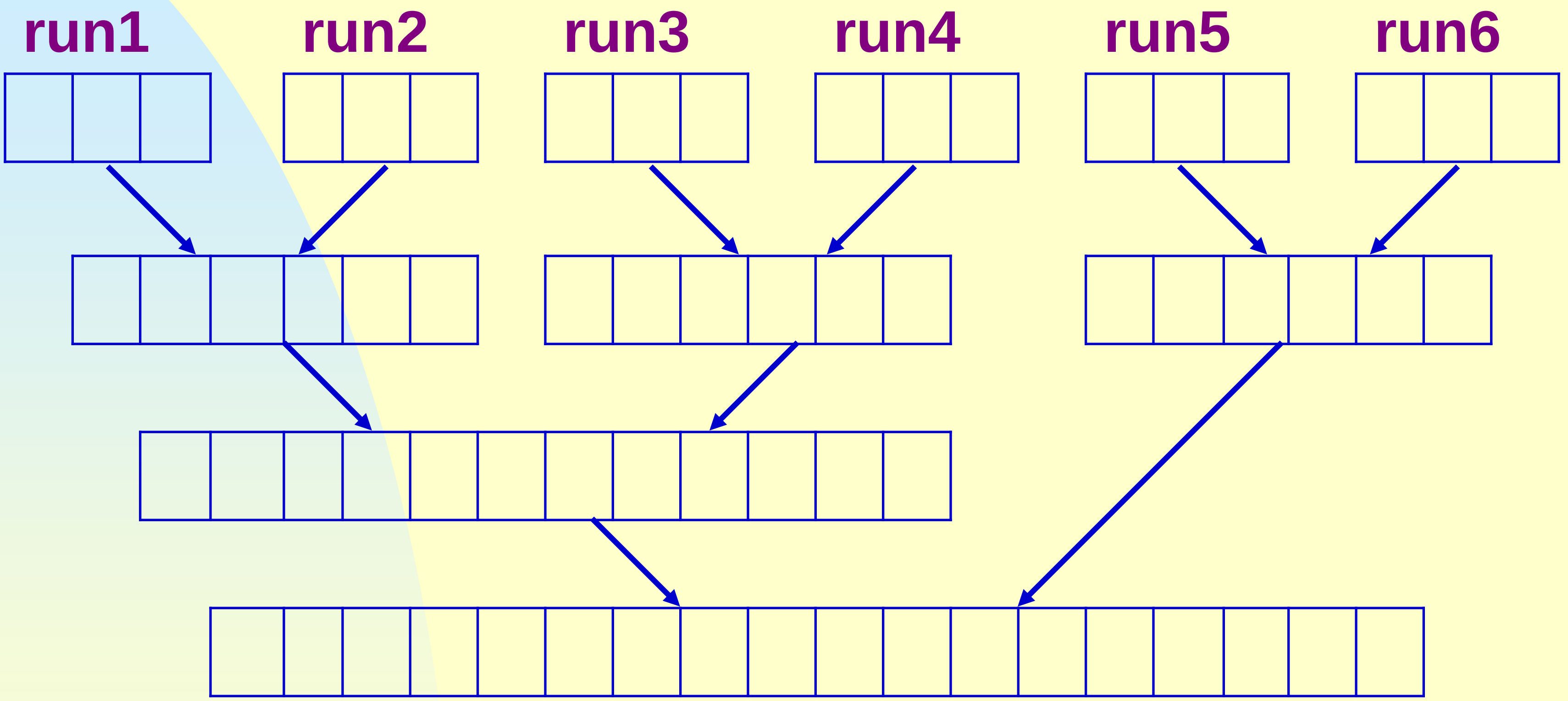
- **block length --- 250 records.**

Now we can sort the list as the following:

(1) Internally sort 3 blocks at a time to obtain 6 runs R_1 to R_6 .



(2) Merge the 6 runs.



Let

t_s = maximum seek time

t_l = maximum latency time

t_{rw} = time to read or write one block of 250 records

$t_{IO} = t_s + t_l + t_{rw}$

t_{IS} = time to internally sort 750 records

$n\ t_m$ = time to merge n records from input buffers to output buffer

The computing times are as:

operation	time
(1) read 18 blocks, internally sort, write 18 blocks	$36\ t_{IO} + 6t_{IS}$
(2) merge runs 1 to 6 in pairs	$36\ t_{IO} + 4500t_m$
(3) merge 2 runs of 1500 records each	$24\ t_{IO} + 3000t_m$
(4) merge 1 run of 3000 records with 1run of 1500	$36\ t_{IO} + 4500t_m$
total time	$132\ t_{IO} + 12000t_m + 6t_{IS}$

The computing time depends chiefly on the number of passes over the data.

We are mainly concerned with:

(1) Reduction of the number of passes by using a high-order merge.

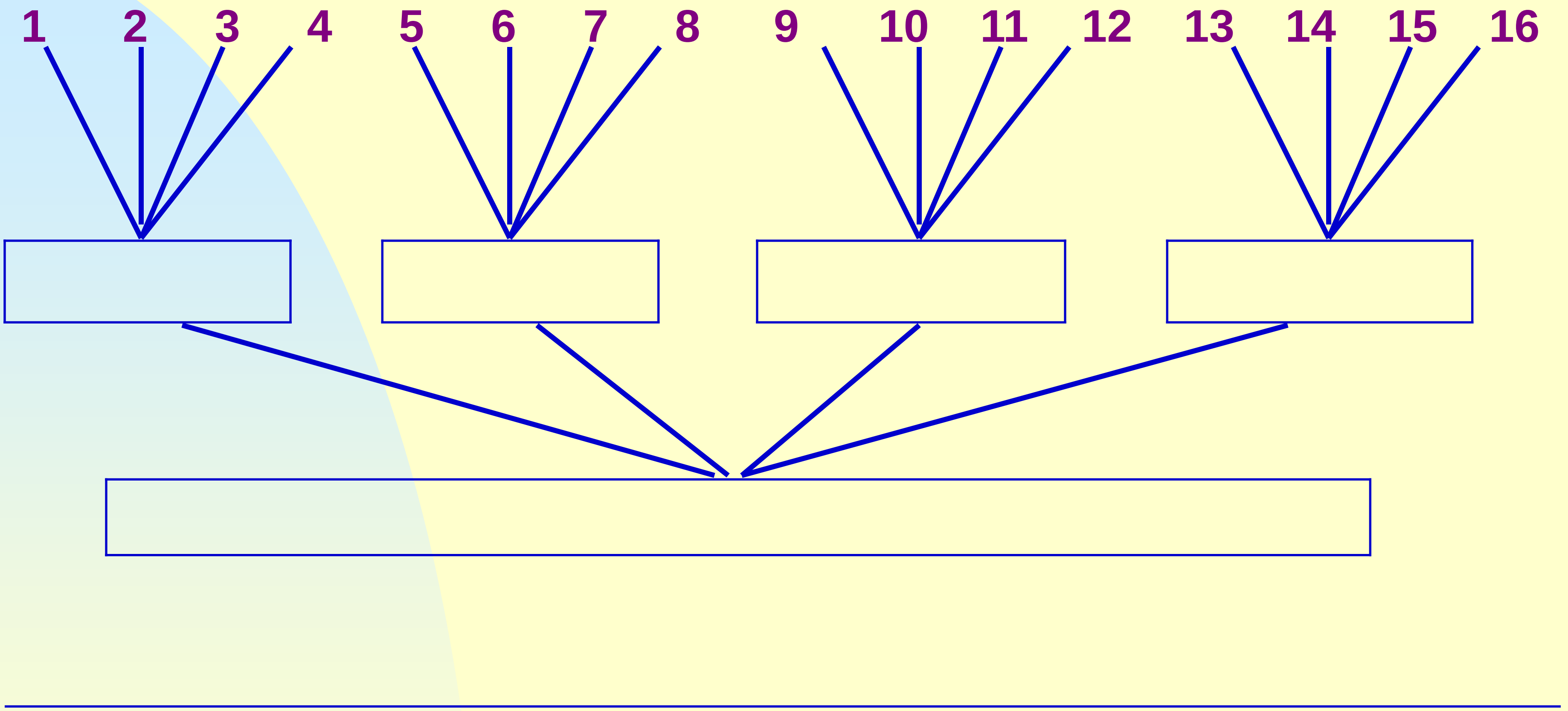
(2) An appropriate buffer-handling scheme to provide for parallel input, output and merging.

(3) Generating fewer runs (or equivalently longer) under the memory limitation.

7.10.2 k-way Merging

For 2-way merge with m runs, a total of $\lceil \log_2 m \rceil$ passes (excluding the one for generating the runs) are needed.

In general, a k-way merge on m runs requires at most $\lceil \log_k m \rceil$ passes over the data. Thus, I/O time may be reduced. As shown in the following tree:


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

For large k (say, $k \geq 6$), to select the smallest from the k possibilities, we can use a loser tree with k leaves.

The total time needed per level of the merge tree is $O(n \log_2 k)$.

The merge tree has $O(\log_k m)$ levels, the total internal merge time is

$$O(n \log_2 k \log_k m) = O(n \log_2 m)$$

which is independent of k!

Comments on high order merge:

(1) save I/O

(2) no big loss of internal processing speed.

(3) the decrease in I/O < indicated by reduction to $\log_k m$ passes, because:

• to do k-way merge we need at least k input buffers and 1 output buffer (or the best totally 2k+2).

• given fixed internal memory,

k increases => smaller buffer size => more blocks => increased seek and latency time.

• hence, beyond a certain k, I/O time increases despite the decrease in the number of passes.

7.10.4 Run Generation

Using a tree of loser, we can generate runs that are , on average, twice as long as obtainable by conventional internal sorting methods.

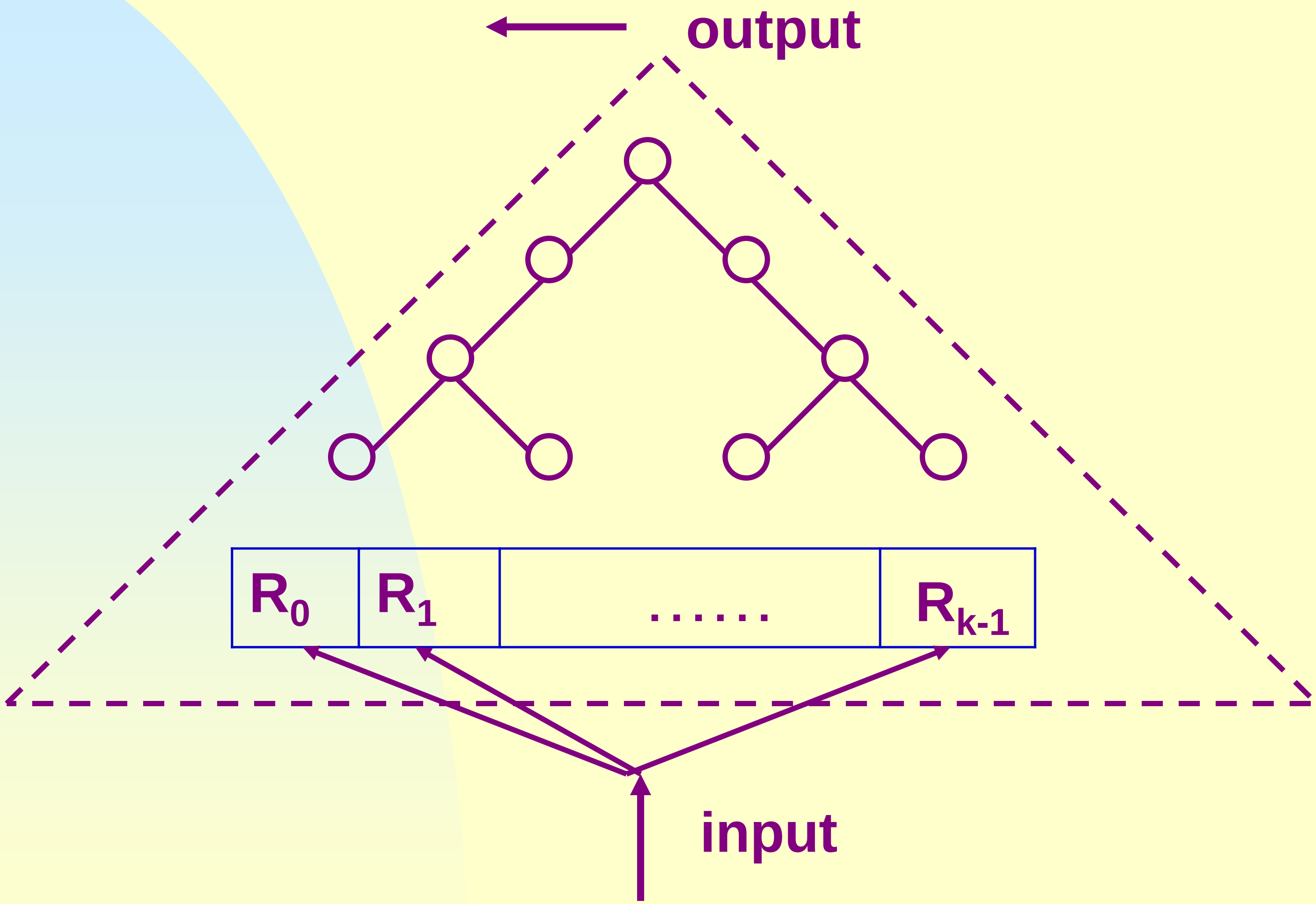
Background: given k record buffers

- **conventional method --- static**

Input list	R_0	R_1	……	R_{k-1}

after finishing sorting on (R_0,…, R_{k-1}), generate k record long runs.

- **tree of loser method --- dynamic**



After one winner is selected, output it, and get a new record from the input list into the winner's buffer →longer runs, on average, 2k long.

Ideas:

- k record buffers r[i], $0 \le i < k$, as leaf nodes, node number=k+i
- k-1 non-leaf nodes numbered 1 to k-1, each non-leaf node i has only one field l[i], $1 \le i < k$, represents the loser of the tournament. l[i] contains the index of the loser's buffer.

- **each r[i] has a run number field rn[i] associated with it to determine whether or not r[i] can be output as part of the run currently generated.**
- **rmax --- the max of the run number of the real records currently in memory.**
- **if input is empty, create a virtual record with rn=rmax+1, to push real records output before it.**

Variable specification:

- **r[i], 0≤i<k --- the k records in the tree of loser**

- **l[i], 1≤i<k --- loser of the tournament played at node i**
- **l[0], q --- winner of the tournament**
- **rn[i], 0≤i<k --- the run number to which r[i] belongs**
- **rc --- current run number**
- **rq --- run number of r[q]**
- **rmax --- number of runs that will be generated**
- **lastRec --- last record output**

```
template <class T>
void Runs (T *r )
{
   r = new T[k];
   int *rn=new int[k], *l=new int[k];
   for (int i = 0; i < k; i++ )  {  //input records
      ReadRecord(r[i]);  rn[i] = 1;
   }
   InitializeLoserTree();
   int q = l[0];   // tournament winner
   int rq=1, rc=1, rmax=1; T LastRec;
```

```
11  while (1) {          // output runs
12      if (rq != rc) {    // end of run
13          output end of run marker;
14          if (rq > rmax) break;  //meet virtual record, to line 35
15          else rc = rq;
16      }
17      WriteRecord(r[q]);  LastRec=r[q];
18      //input new record into tree
19      if (end of input) rn[q]=rmax+1; //generate virtual record
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      }
```

```
      rq=rn[q];
      // reconstruct the tree of losers
     for (int t=(k+q)/2;t; t/=2) // t is initialized to be parent of q
        if (rn[l[t]]<rq || rn[l[t]]==rq && r[l[t]]<r[q])
        {  // t is the winner
           swap(q, l[t]);
           rq = rn[q];
        }
   }  // end of  while
   delete [ ] r; delete [ ] rn;  delete [ ] l;
}
```

Analysis of runs:

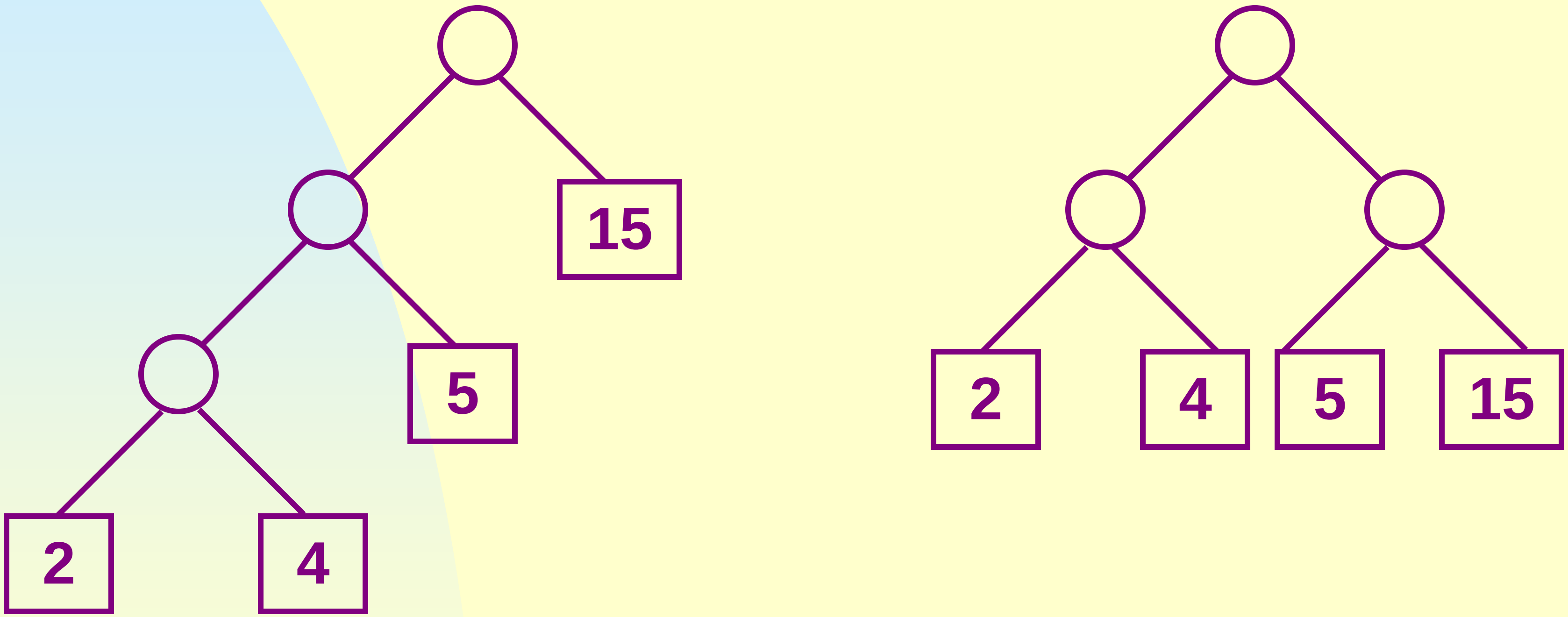
When the input list is sorted, only one run is generated. On average, the run size is 2k. The time required to generate all runs for an n record list is O(n log k).

7.10.5 Optimal Merging of Runs

When runs are of different size, the merging strategy of making complete passes over all runs does not yield minimum run times.

For example, suppose we have 4 runs of length 2, 4, 5, and 15, respectively.

2 merge trees for merging the 4 runs:



(a)

(b)

- **internal nodes** --- the circular nodes.
- **external nodes** --- the square nodes.

Given n runs, there are n external nodes in the merge tree. Let the external nodes numbered 1 to n, q_i be the size of the run node i represents, d_i be the distance from the root to the external node i, $1 \le i \le n$. Then the total merge time is

$$E_w = \sum_{1 \le i \le n} q_i d_i$$

The E_w is called the **weighted external path length.**

The E_w

- **of (a) is 2*3 + 4*3 + 5*2 + 15*1 = 43**
- **of (b) is 2*2 + 4*2 + 5*2 + 15*2 = 52**

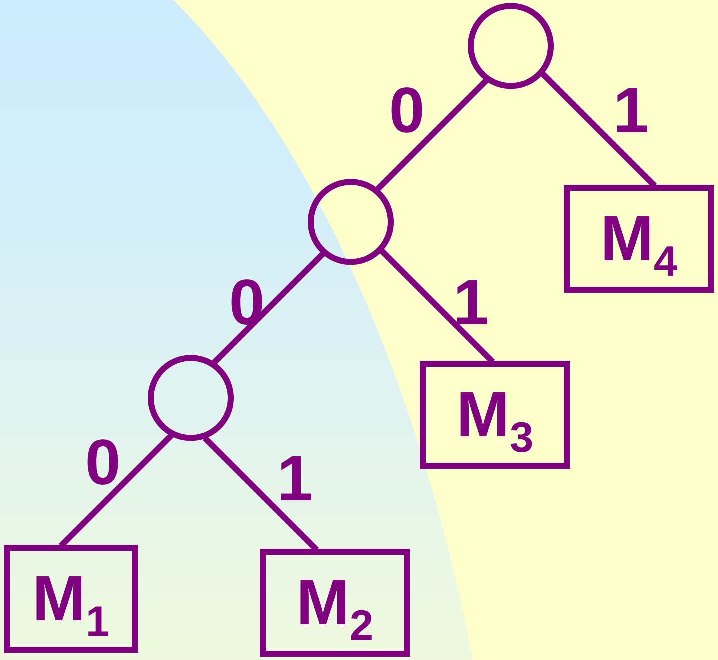
The cost of a k-way merge n runs of length q_i, $1 \le i \le n$, is minimized by using a merge tree of degree k that has minimum E_w.

We shall consider k=2 only. The result is easily generalized to the case k>2.

Another application for binary tree with minimum Ew is to obtain an optimal set of codes for messages $M_1,\ldots,M_{n+1}$.

Each code is a binary string that will be used for transmission of the corresponding message.

These codes are called Huffman codes.



q_i--- relative frequency for M_i.

d_i--- distance from the root to the external node for M_i.

The cost of decoding is

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

Basic ideas of Huffman's solution :

- begin with a min heap of n single-node trees, which is an external node with the weight being one of the q_i's.
- repeatedly extract 2 minimum-weight **a** and **b** from the min heap, combine them into a single binary tree **c**, and insert c into the min heap.
- after n-1 rounds, the min heap will be left with a single binary tree, which is the wanted.

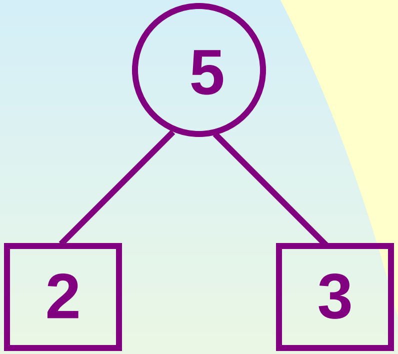
Assume:

- **the function Huffman is a friend of TreeNode.**
- **use the data field of a TreeNode to store the weight of the binary tree rooted at that node.**

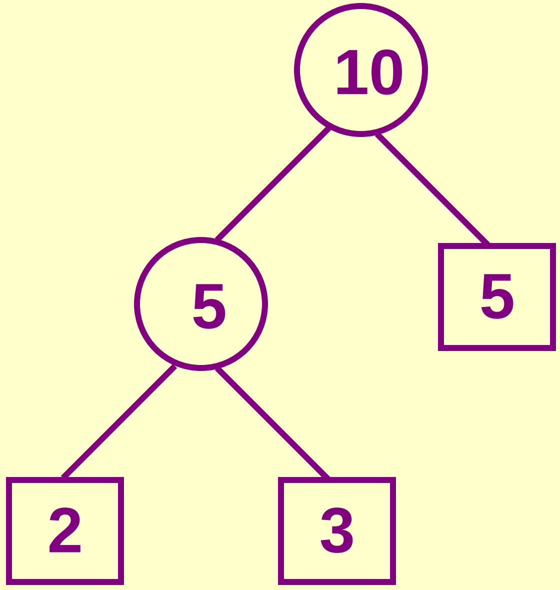
```
template <class T>
void Huffman(MinHeap<TreeNode<T> *> heap, int n)
{ // heap is initially a min heap of n single-node binary trees.
    for (int i=0; i<n-1; i++) {//loop n-1 times to combine n nodes
      TreeNode<T> *a=heap.Top();
      heap.Pop();
      TreeNode<T> *b=heap.Top();
       heap.Pop();
      TreeNode<T> *c=new TreeNode(a→data+b→data,
                                     a, b);  // refer section 5.2.3.2
       heap.Push(c);
    }
}
```

Example 7.15:

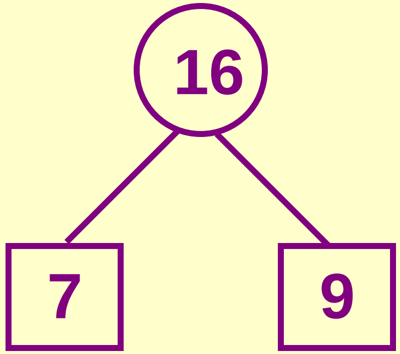
Suppose we have the weights q_1=2, q_2=3, q_3=5, q_4=7, q_5=9, and q_6=13, then the sequence of trees we get is as the following:



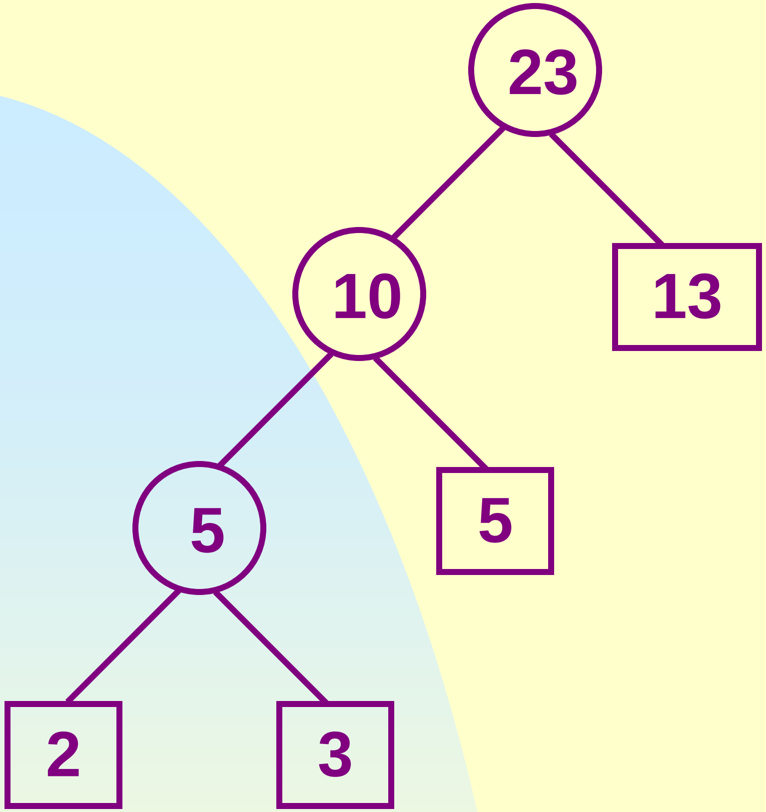
(a) q_1+q_2



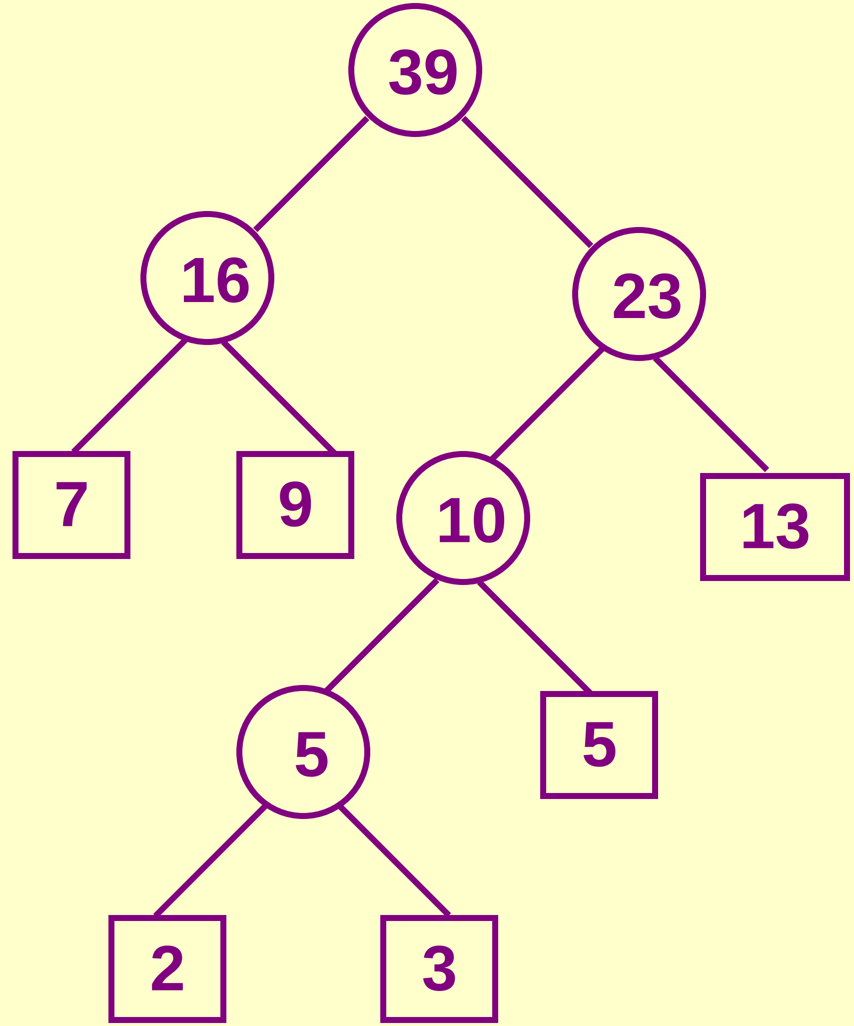
(b) (a)+q_3



(c) q_4+q_5



(d) (b)+q_6



(e) (c)+(d)

Analysis of huffman:

- the main loop --- n-1 times
- each call to Top is O(1), to Pop and Push is O(log n)

The total time: O(n log n).

Exercises: P457-2