Data Structures: Sorting

Wei-Mei Chen

Department of Electronic and Computer Engineering
National Taiwan University of Science and Technology

Introduction

Motivation

- Why efficient sorting methods are so important?
- The efficiency of a searching strategy depends on the assumptions we make about the arrangement of records in the list
- No single sorting technique is the “best” for all initial orderings and sizes of the list being sorted.
- We examine several techniques, indicating when one is superior to the others.
- Assumption
  The term **list** here is a collection of records. Each record has one or more fields. Each record has a key to distinguish one record with another
  - ☞ name or phone number.

Sequential Search

- We search the list by examining the key values

  Example: 4, 15, 17, 26, 30, 46, 48, 56, 58, 82, 90, 95

```
int seqSearch(element a[], int k, int n)
{   int i;    /* data → a[1:n]
    for(i=1; i<=n && a[i].key!=k; i++) ;
    if (i>n) return 0;
    return i; /* return i if a[i]==key; otherwise return 0
}
```

- Analysis:
  - Unsuccessful search: $n+1 \Rightarrow O(n)$
  - Average successful search:

$$\frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2} = O(n)$$

Binary Search

- Consider a sorted list: $\{8, 23, 32, 45, 56, 78\}$
  Figure out if *searchnum* is in the list.
  - YES ➠ i where list[i]=*searchnum*
  - NO ➠ -1

```
while (more integers to check) {
  middle = (left + right) / 2;
  if (searchnum < list[middle])
    right = middle - 1;
  else if (searchnum == list[middle])
    return middle;
  else left = middle + 1;
}
```

Interpolation Search

- Consider to look for the word "ZOO" in a dictionary
  We won't start the search at the middle.
- Comparing k with $a[i].key$

$$i = \frac{k - a[1].key}{a[n].key - a[1].key} \times n$$

Interpolation search can be used only when the file is ordered.

List Verification

- Assume that **`list1[1..n]`** and **`list2[1..m]`**. Determine if
    1. For a record with key **`list1[i].key`**, there is no record with the same key in **`list2`**
    2. If **`list1[i].key==list2[j].key`**, the two records do not match on at least one of **the other fields**

Verifying Two Unsorted Lists

```
void verify1(element list1[], element list2[], int n, int m)
{ /* Compare list1[1:n] and list2[1:m] */
   int i, j, marked[MAX_SIZE];
   for (i=1; i<=m; i++)
      marked[i] = FALSE;
   for (i=1; i<=n; i++)
      if(( j = seqSearch(list2, m, list1[i].key)) == 0)
         printf("%d is not in list 2\n", list1[i].key);
      else
         marked[j] = TRUE;
   for (i=1; i<=m; i++)
      if(!marked[i])
         printf("%d is not in list 1\n", list2[i].key);
}
```

Worst-case complexity: $O(mn)$

Verifying Two Sorted Lists

```
void verify2(element list1[], element list2[], int n, int m)
{   int i, j;
    sort(list1, n); sort(list2, m);
    i = j = 1;
    while( i<=n && j<=m)
        if (list1[i].key < list2[j].key) {
            printf("%d is not in list 2\n", list1[i].key); i++;
        }
        else if (list1[i].key == list2[j].key) {
            i++; j++;
        }
        else {
            printf("%d is not in list 1\n", list2[j].key); j++;
        }
    for (; i<=n; i++)   printf("%d is not in list 2\n", list1[i].key);
    for (; j<=m; j++)   printf("%d is not in list 1\n", list2[j].key);
}
```

$$O(t_{Sort}(n) + t_{Sort}(m) + n + m)$$

➠ Worst-case complexity: $O(\max\{n \log n, m \log m\})$

Definitions

- Given a list of records $(R_1, R_2, \ldots, R_n)$. Each record has a key K_i. The sorting problem is then that of finding permutation, σ, such that $K_{\sigma(i)} \leq K_{\sigma(i+1)}, 1 \leq i \leq n-1$. The desired ordering is $(R_{\sigma(1)}, R_{\sigma(2)}, \ldots, R_{\sigma(n)})$.
- If a list has several key values that are identical, the permutation, σ_s, is not unique. Let σ_s be the permutation of the following properties:
    1. $K_{\sigma(i)} \leq K_{\sigma(i+1)}, 1 \leq i \leq n-1$.
    2. If $i < j$ and $K_i == K_j$ in the input list, then R_i precedes R_j in the sorted list.

  ➠ **stable**

An Example for Stable

Destination	Airline	Flight	Sched
Buffalo	Air Tran	549	10:42 AM
Atlanta	Delta	1097	11:00 AM
Baltimore	Southwest	836	11:05 AM
Atlanta	Air Tran	872	11:15 AM
Atlanta	Delta	28	12:00 PM
Boston	Delta	1056	12:05 PM
Baltimore	Southwest	216	12:20 PM
Austin	Southwest	1045	1:05 PM
Albany	Southwest	482	1:20 PM
Boston	Air Tran	515	1:21 PM
Baltimore	Southwest	272	1:40 PM
Atlanta	Alltalia	3429	1:50 PM

Destination	Airline	Flight	Sched
Albany	Southwest	482	1:20 PM
Atlanta	Delta	1097	11:00 AM
Atlanta	Air Tran	872	11:15 AM
Atlanta	Delta	28	12:00 PM
Atlanta	Alltalia	3429	1:50 PM
Austin	Southwest	1045	1:05 PM
Baltimore	Southwest	836	11:05 AM
Baltimore	Southwest	216	12:20 PM
Baltimore	Southwest	272	1:40 PM
Boston	Delta	1056	12:05 PM
Boston	Air Tran	515	1:21 PM
Buffalo	Air Tran	549	10:42 AM

1. **Schedule**
2. **Destination**

Categories of Sorting Method

- **Internal Method**: Methods to be used when the list to be sorted is small enough so that the entire sort list can be carried out in the main memory.
  - ☞ Insertion sort, quick sort, merge sort, heap sort and radix sort.
- **External Method**: Methods to be used on larger lists.

Insertion Sort

Insertion Sort

```
void insert(element e, element a[], int i)
{ /* a[1:i]→ a[1:i+1] sorted
     a[0] = e;
     while (e.key < a[i].key) {
          a[i+1] = a[i];
          i--;
     }
     a[i+1] = e;
}
```

```
void insertionSort(element a[], int n)
{ /* a[1:n] → sorted
     int j;
     for ( j =2; j<=n; j++) {
          element temp = a[j];
          insert(temp, a, j-1);
     }
}
```

5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
2	4	5	6	1	3
1	2	4	5	6	3
1	2	3	4	5	6

Loop invariant : At the start of each iteration of the for loop, $A[1..j-1]$ contains it's original elements but sorted

Analysis of Insertion Sort

- In the worst case, $insert(e, a, i)$ makes $i + 1$ comparisons before making the insertion.

  Thus, the total cost is $O\left(\sum_{i=1}^{n-1}(i+1)\right) = O(n^2)$
- Variations
  - Binary insertion sort:
    the number of comparisons in an insertion sort can be reduced if we replace the sequential search by binary search. The number of records moves remains the same.
  - List insertion sort:
    The elements of the list are represented as a linked list rather than an array. The number of record moves becomes zero.

Quick Sort

Quick Sort

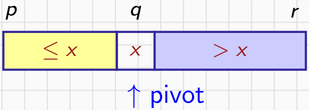
Hoare [1962]

- the best practical sorting algorithm
- in place sorting
- worst-case running time: $\Theta(n^2)$ on n numbers
- average-case running time: $\Theta(n \lg n)$

Description of Quick Sort

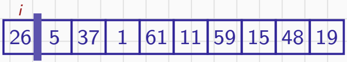
Divide-and-conquer paradigm

- Divide: $a[p..r]$

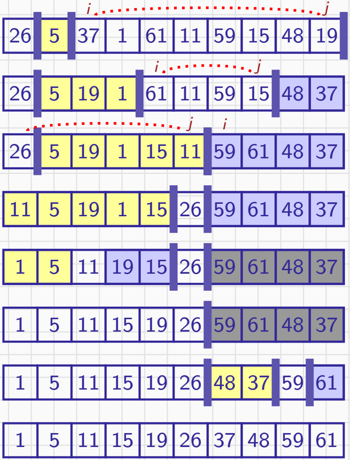


- Conquer:
  Sort the two subarrays $a[p..q-1]$ and $a[q+1..r]$ by recursive calls to quick sort.
- Combine:
  Since the subarrays are sorted in place, no work is needed to combine them

Algorithm quickSort



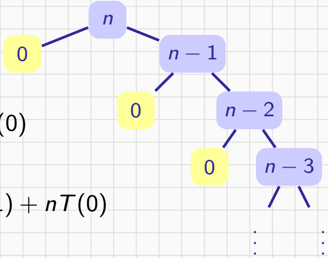

```
void quickSort(element a[],int left,int right)
{ /* a[left].key:  the pivot key */
    int pivot, i, j;
    element temp;
    if (left < right) {
        i = left; j = right+1;
        pivot = a[left].key;
        do { /*partition by swapping*/
            do i++; while(a[i].key < pivot);
            do j--; while(a[j].key > pivot);
            if (i<j) SWAP(a[i], a[j], temp);
        }while (i<j);
        SWAP( a[left], a[j], temp);
        quickSort(a, left, j-1);
        quickSort(a, j+1, right);
    }
}
```



Worst-Case Partitioning

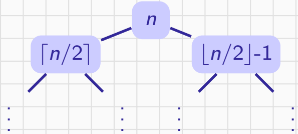
- The worst-case behavior for quick sort occurs when the partitioning routine produces one subproblem with $n-1$ elements and one with 0 elements.
- Assume $T(0) = \Theta(1)$,

$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + n - 1 \\
&= T(n-1) + n - 1 + T(0) \\
&= T(n-2) + (n-2) + (n-1) + 2T(0) \\
&\cdots \\
&= T(0) + 0 + 1 + 2 + 3 + \cdots + (n-1) + nT(0) \\
&= \Theta(n^2)
\end{aligned}
$$



Best-Case Partitioning

- In the most even possible split, PARTITION produces two subproblems.



- Then $T(n) \leq 2T(n/2) + \Theta(n)$
  $\Rightarrow$ we have $T(n) = O(n \lg n)$
- Thus the equal balancing of the two sides of the partition at every level of the recursion produces an asymptotically faster algorithm.

Average-Case Running Time

In the average case, PARTITION produces a mix of "good" and "bad" splits.

- Consider
  Basic operation: the comparison of $a[i]$ with a pivot $a[r]$ in PARTITION.
  Input size: n, the number of items in a
- Assume that the value of pivot is equally likely to be any of the numbers from 1 through n.

$$T(n) = \sum_{p=1}^{n} \underset{\uparrow \text{ Probability}}{\frac{1}{n}} [T(p-1) + T(n-p)] + \overbrace{n-1}^{\text{Time to partition}}$$

Since $\sum_{p=1}^{n} \frac{1}{n}\left[T(p-1)+T(n-p)\right] = \frac{2}{n}\sum_{p=1}^{n} T(p-1)$, we have

$$T(n) = \frac{2}{n}\sum_{p=1}^{n} T(p-1) + n - 1.$$

$$\text{"}\times n\text{"}: \qquad nT(n) = 2\sum_{p=1}^{n} T(p-1) + n(n-1) \qquad (1)$$

$$\text{"}n \to n-1\text{"}: (n-1)T(n-1) = 2\sum_{p=1}^{n-1} T(p-1) + (n-1)(n-2) \qquad (2)$$

$$\text{"}(1)-(2)\text{"} \Rightarrow nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1)$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2(n-2)}{(n-1)(n)}$$

$$\vdots \qquad \vdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2}{6}$$

$$\frac{n-1}{n(n+1)} = \frac{2}{n+1} - \frac{1}{n}$$

$$\frac{n-2}{(n-1)n} = \frac{2}{n} - \frac{1}{n-1}$$

summing $\Rightarrow \dfrac{T(n)}{n+1} = \dfrac{4}{n+1} + 2\left(\dfrac{1}{n} + \dfrac{1}{n-1} + \cdots + \dfrac{1}{3}\right) - 1$

$$= 2\sum_{k=3}^{n} \frac{1}{k} + \frac{4}{n+1} - 1$$

$$= 2H_n + \frac{4}{n+1} - 4$$

$$\boxed{T(n) \approx 2(n+1)\ln n \in \Theta(n \lg n)}$$

Harmonic Numbers

$$
\begin{aligned}
H_n &= \sum_{k=1}^{n} \frac{1}{k} \\
&= 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \\
&= \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \frac{1}{252n^6} + \cdots
\end{aligned}
$$

where $\gamma = 0.5772156649$

Variations of Quick Sort

1. Median of three:
   Pick the median of the first, middle, and last keys in the current sublist as the pivot. Thus,
   $$\text{pivot} = \textit{median}\{K_l, K_{(l+r)/2}, K_r\}.$$
   ➠ Use *median* as the pivot
2. Randomized Quick Sort:
   Pick a key randomly.

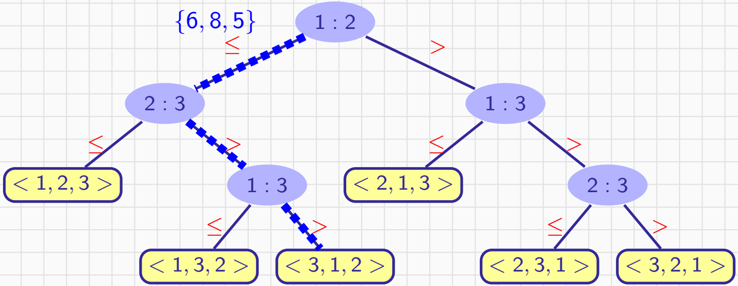
Optimal Sorting Time

How Fast Can We Sort?

- We will provide a lower bound, then beat it. How do you suppose we'll beat it?
- All of the sorting algorithms so far are **comparison sorts**.
  - Basic operation: a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are $\Omega(n \lg n)$

Decision Trees

- Decision trees provide an abstraction of comparison sorts
- A decision tree represents the comparisons made by a comparison sort.
- The decision tree for insertion sort operating on three elements.



There are $3! = 6$ possible permutations (leaves)

Lower Bounds

- A lower bound of a problem is the least time complexity required for any algorithm which can be used to solve this problem.
  - worst case lower bound (lower bound)
  - average case lower bound
- The lower bound for a problem is **not unique**.

  ☞ as tight as possible
  - e.g. $\Omega(1), \Omega(n), \Omega(n \log n)$ are all lower bounds for sorting.
  - $\Omega(1), \Omega(n)$ are trivial
- If the present lower bound is $\Omega(n \log n)$ and there is an algorithm with time complexity $O(n \log n)$, then the algorithm is **optimal**.

Lower Bound for Sorting

- The length of the longest path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs.
  ➠ the worst-case #of comparisons
  = the height of its decision tree.
- The worst-case time complexity:
  the longest path from the top of the tree to a leaf node
- Balanced tree has the smallest depth:

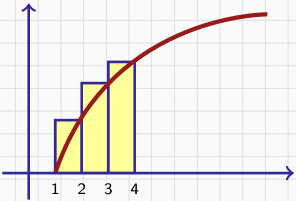
$$\lceil \log(n!) \rceil = \Omega(n \log n)$$

➠ lower bound for sorting: $\Omega(n \log n)$

Theorem

Any algorithm that sorts only by comparisons must have a worst-case computing time of $\Omega(n \log n)$

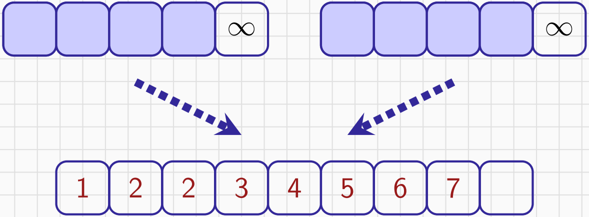
$$
\begin{aligned}
\log(n!) &= \log(n(n-1)(n-2)\cdots 1) \\
&= \log 2 + \log 3 + \cdots + \log n \\
&> \int_1^n \log x dx \\
&= \log e \int_1^n \ln x dx \\
&= \log e \left[x \ln x - x \right]_1^n \\
&= \log e \left[n \ln n - n + 1 \right] \\
&= n \log n - n \log e + 1.44 \\
&\geq n \log n - 1.44n \\
&= \Omega(n \log n)
\end{aligned}
$$



Merge Sort

Example: Merging

Merge two sorted lists into one sorted list

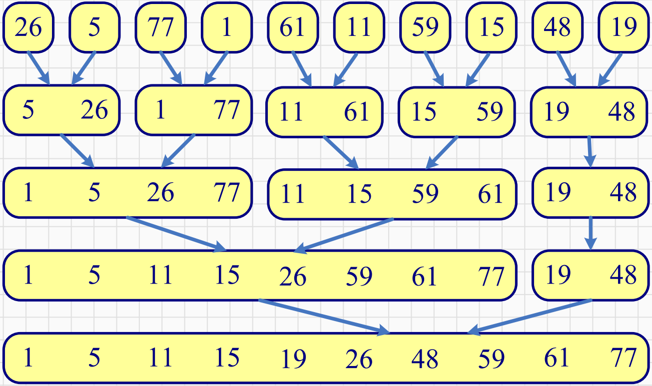


Merging Two Sorted Lists

```
void merge(element initList[], element mergedList[], int i, int m, int n)
{/* initList[i:m] and initList[m+1:n] ➜ mergedList[i:m]*/
    int j, k, t;
    j = m+1;
    k = i;
    while( i<=m && j<=n) {
        if (initList[i].key <= initList[j].key)
            mergedList[k++] = initList[i++];
        else
            mergedList[k++] = initList[j++];
    }
    if(i>m)/* mergedList[k:n] = initList[j:n] */
        for (t = j; t<=n; t++)
            mergedList[t] = initList[t];
    else /* mergedList[k:n] = initList[i:m] */
        for (t = i; t<=m; t++)
            mergedList[k+t-i] = initList[t];
}
```

$\boxed{O(n-i+1)}$

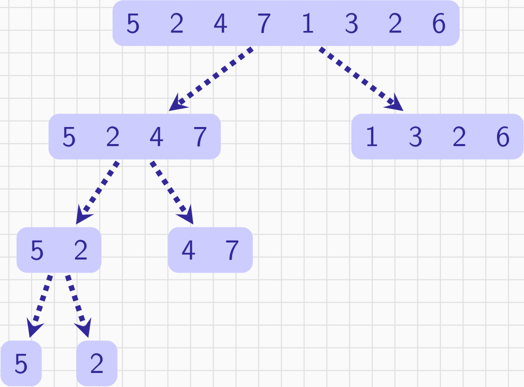
Iterative Merge Sort



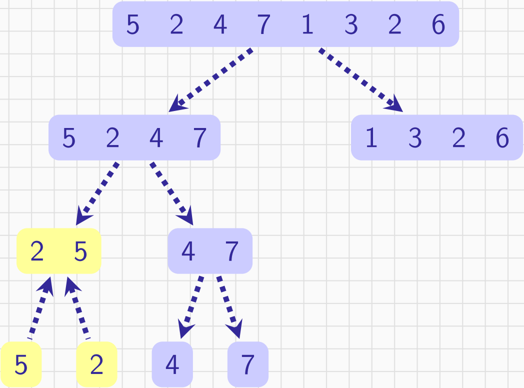
```c
void mergePass(element initList[], element mergedList[], int n, int s)
{
    int i, j;
    for( i=1; i<= n - 2 * s + 1; i+= 2*s)
        merge(initList, mergedList, i, i+s-1, i+2*s-1);
    if ( i+s-1 < n)
        merge(initList, mergedList, i, i+s-1, n);
    else
        for( j=i; j<=n; j++)
            mergedList[j] = initList[j];
}
```

```c
void mergeSort(element a[], int n)
{ /* sort a[1:n] */
    int s =1;
    element extra[MAX_SIZE];
    while(s<n) {
        mergePass(a, extra, n, s);
        s*=2;
        mergePass(extra, a, n, s);
        s*=2;
    }
}
```

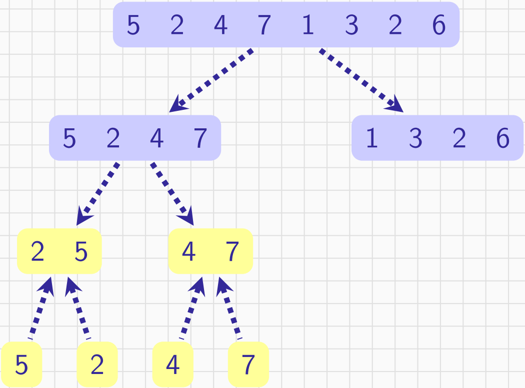
Example: Recursive Merge Sort



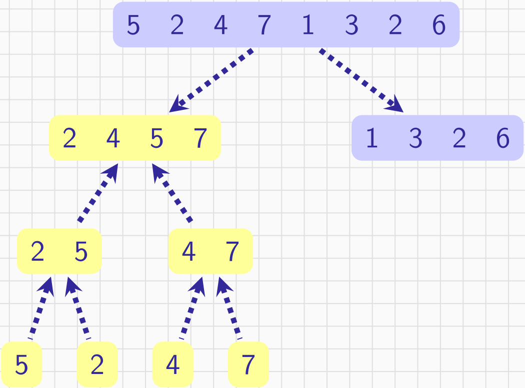
Example: Recursive Merge Sort



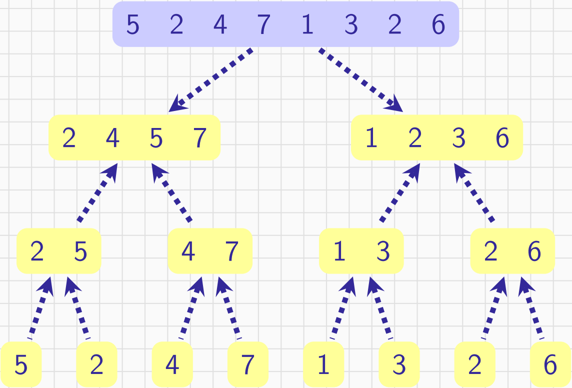
Example: Recursive Merge Sort



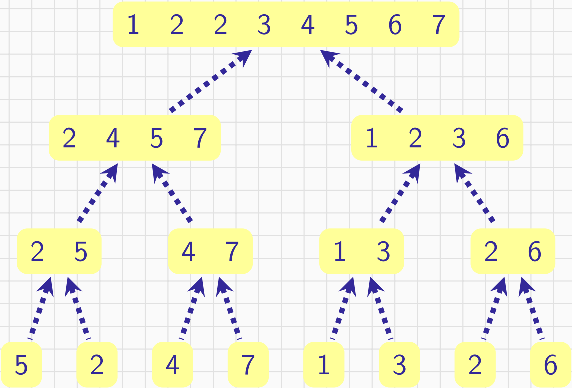
Example: Recursive Merge Sort



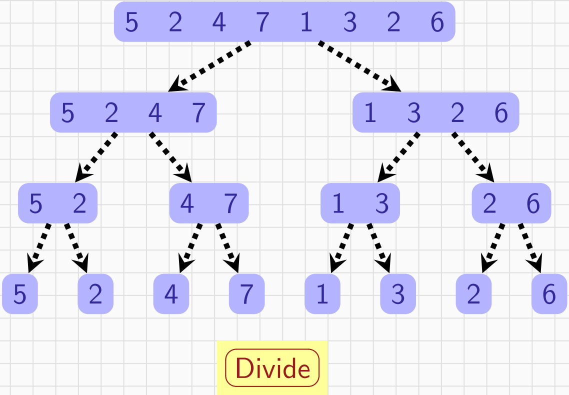
Example: Recursive Merge Sort



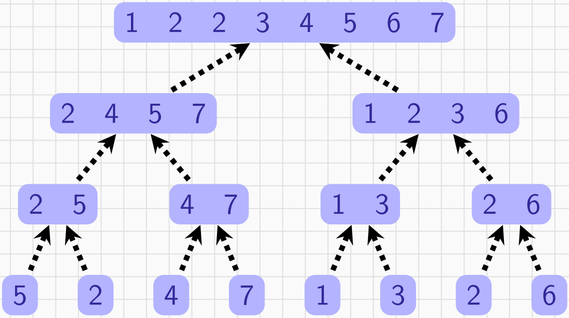
Example: Recursive Merge Sort



Dividing Phase

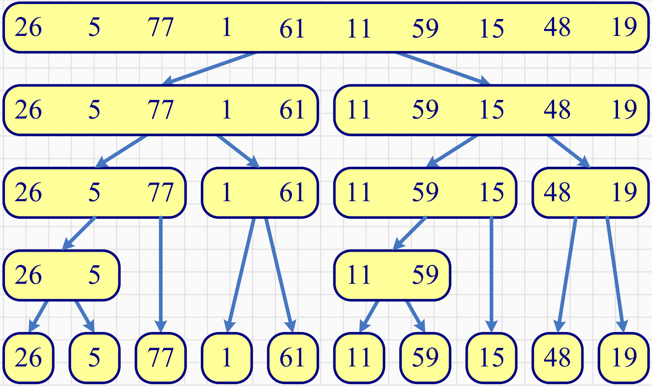


Merging Phase

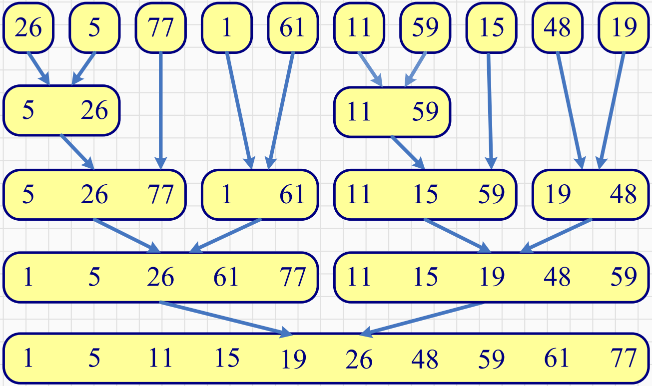


Conquer and Combine

Dividing Phase: Recursive Merge Sort



Merging Phase: Recursive Merge Sort



```
int listMerge(element a[], int link[], int start1, int start2)
{   int last1, last2, lastResult=0;
    for(last1 = start1, last2 = start2; last1 && last2; )
        if (a[last1] <= a[last2]) {
            link[lastResult] = last1;
            lastResult = last1;
            last1 = link[last1];
        }
        else{
            link[lastResult2] = last2;
            lastResult = last2;
            last2 = link[last2];
        }
    if(last1 == 0) link[lastResult] = last2;
    else link[lastResult] = last1;
    return link[0];
}
```

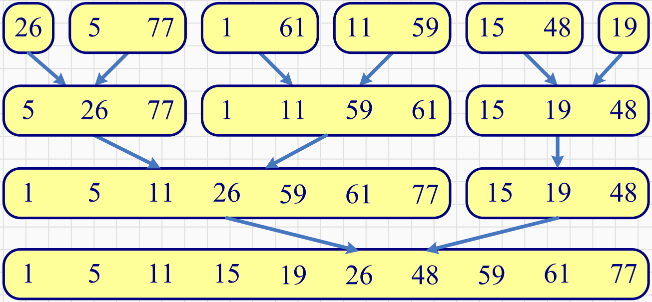
```
int rmergeSort(element a[], int link[], int left, int right)
{   if(left >= right) return left;
    int mid = (left + right)/2;
      return listMerge(a, link,rmergeSort(a, link, left, mid),
                                    rmergeSort(a, link, mid+1, right));
}
```

Analysis of Merge Sort

- Time Complexity:

$$
\begin{aligned}
T(n) &= \begin{cases} O(1), & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases} \\
&= O(n \log n)
\end{aligned}
$$

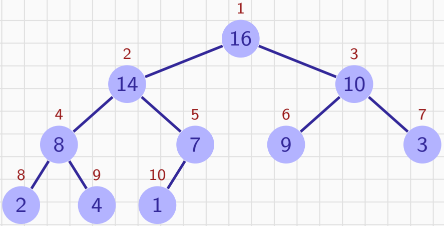
Variations: Natural Merge Sort

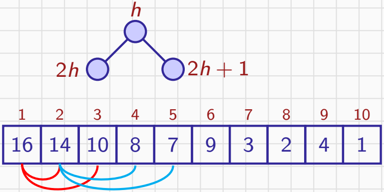


Heap Sort

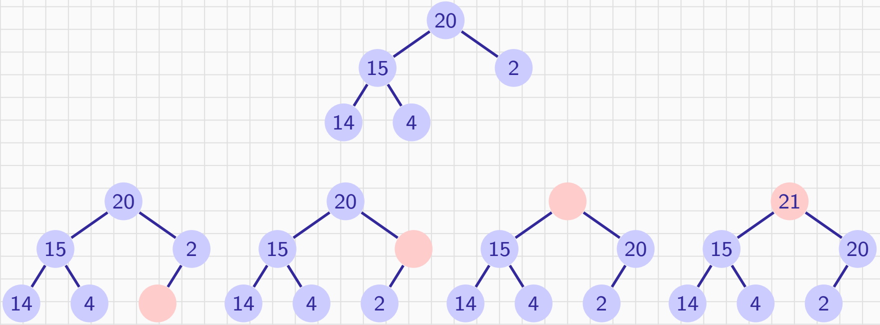
Recall: Heaps

- A max tree is a tree in which the key value in each node is no smaller than the key values in its children. A max heap is a complete binary tree that is also a max tree.

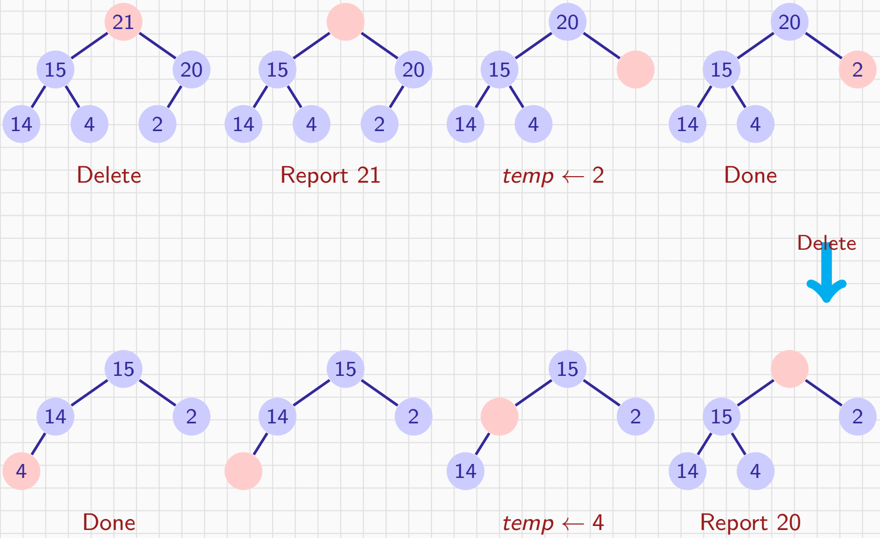




Inserting 21 into a Max Heap



Deleting from a Max Heap

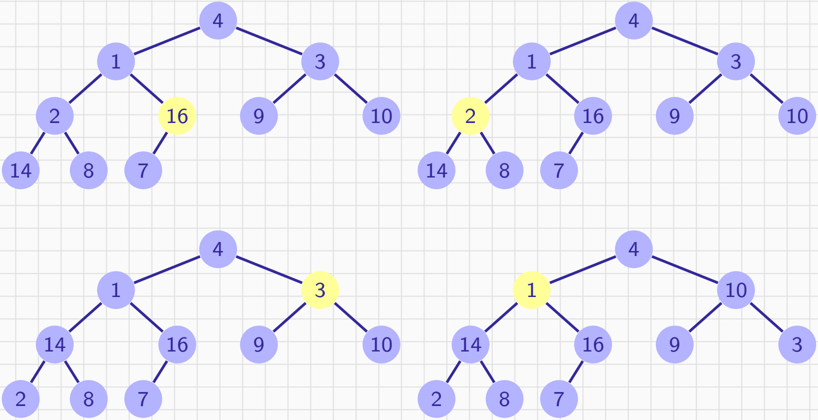


Adjusting a Max Heap

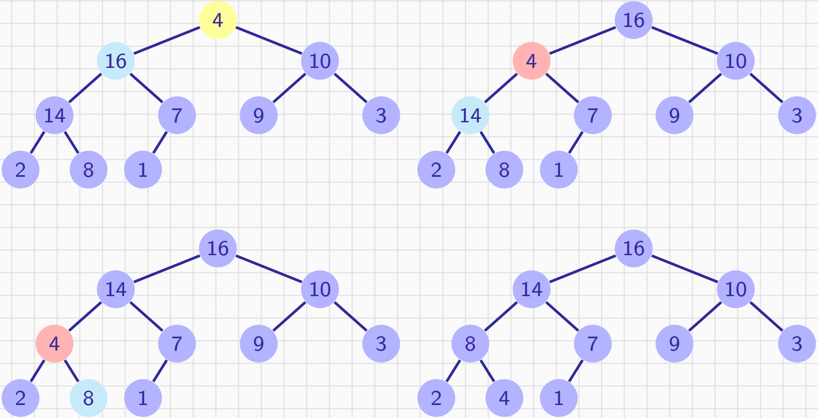
```
void adjust(element a[], int root, int n)
{   int child, rootkey;
    element temp;
    temp = a[root];
    rootkey = a[root].key;
    child = 2*root; /* left subtree */
    while(child <= n) {
        if ((child < n) && (a[child].key < a[child+1].key))
            child++;
        if (rootkey > a[child].key) break;
        else {
            a[child/2] = a[child]; /* move to parent */
            child*=2;
        }
    }
    a[child/2] = temp;
}
```

```
for (i = n/2; i>0; i--)
    adjust(a, i, n);
```
Establish a max heap

Building a Max Heap (1/2)



Building a Max Heap (2/2)



Analysis of `adjust`

$L = \lfloor \lg n \rfloor$: height i: the level of an internal node

➠ # of comparisons is at most:

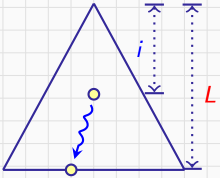
$$\sum_{i=0}^{L-1} 2(L-i)2^i = 2L\sum_{i=0}^{L-1} 2^i - 2\sum_{i=0}^{L-1} i \cdot 2^i$$

$$= 2L(2^L - 1) - 2(2^L(L-2) + 2)$$

$$= 4 \cdot 2^L - 2L - 4$$

$$= 4 \cdot 2^{\lfloor \lg n \rfloor} - 2\lfloor \lg n \rfloor - 4$$

$$\leq cn \quad (\text{ if } c \geq 4)$$



Note: $\sum_{i=0}^{L-1} i \cdot 2^i = 2^L(L-2) + 2$

Compute $\sum_{i=0}^{L-1} i \cdot 2^i$

Let $S = \sum_{i=1}^{L-1} i \cdot 2^i$

$$S = 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + 4 \cdot 2^4 + \cdots + (L-1) \cdot 2^{L-1}$$
$$2S = \quad 2^2 + 2 \cdot 2^3 + 3 \cdot 2^4 + \cdots + (L-2) \cdot 2^{L-1} + (L-1)2^L$$

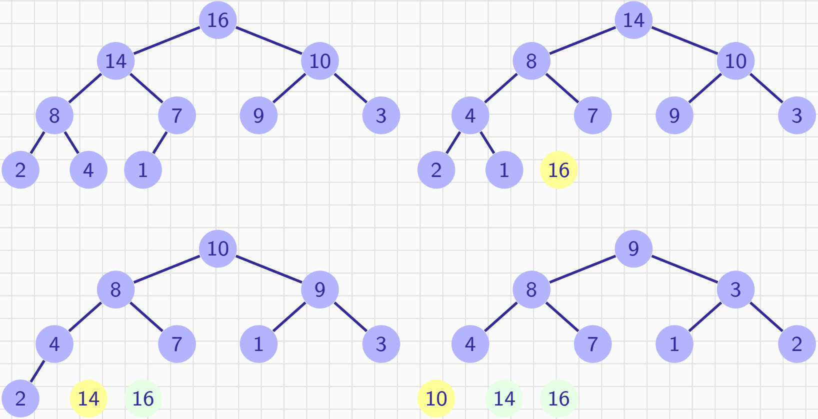
$\Rightarrow$ $$S - 2S = 2 + 2^2 + 2^3 + \cdots + 2^{L-1} - (L-1)2^L$$

$$\begin{aligned} S &= (L-1)2^L - \frac{2(1-2^{L-1})}{1-2} \\ &= 2^L(L-1) + 2 - 2 \cdot 2^{L-1} \\ &= 2^L(L-2) + 2 \end{aligned}$$

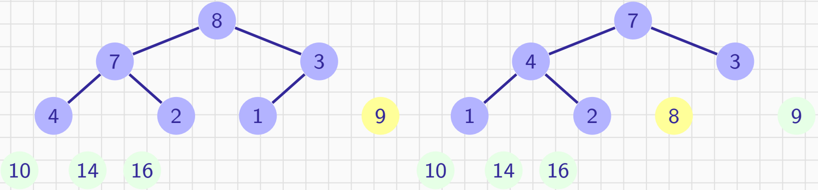
Algorithm heapSort

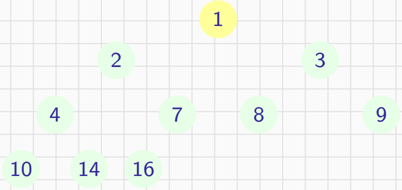
```
void heapSort(element a[], int n)
{ /* sort a[1:n] */
    int i, j;
    element temp;
    for( i = n/2; i>0; i--)
        adjust(a, i, n);
    for( i = n-1; i>0; i--) {
        SWAP(a[1], a[i+1], temp);
        adjust(a, 1, i);
    }
}
```

Example: Heap Sort (1/2)



Example: Heap Sort (2/2)

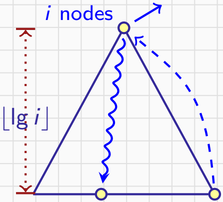




1	2	3	4	7	8	9	10	14	16

Analysis of Heap Sort

$$
\begin{aligned}
&2\sum_{i=1}^{n-1}\lfloor \lg i\rfloor \\
&= 2\left(n\lfloor \lg n\rfloor - 2^{\lfloor \lg n\rfloor+1} + 2\right) \\
&= 2n\lfloor \lg n\rfloor - 4\cdot 2^{\lfloor \lg n\rfloor} + 4 \\
&= 2n\lfloor \lg n\rfloor - 4cn + 4 \text{ if } 2 \le c \le 4
\end{aligned}
$$



Compute $\sum_{i=1}^{n-1} \lfloor \lg i \rfloor$

$$\lfloor \lg 1 \rfloor = 0 \quad \Rightarrow 1$$
$$\lfloor \lg 2 \rfloor = \lfloor \lg 3 \rfloor = 1 \quad \Rightarrow 2$$
$$\lfloor \lg 4 \rfloor = \lfloor \lg 5 \rfloor = \lfloor \lg 6 \rfloor = \lfloor \lg 7 \rfloor = 2 \quad \Rightarrow 4 \qquad ☞\ k \cdot 2^k$$
$$\lfloor \lg 8 \rfloor = \cdots = \lfloor \lg 15 \rfloor = 3 \quad \Rightarrow 8$$

$$\sum_{k=0}^{m-1} k \cdot 2^k = 2^m(m-2) + 2$$

$$\begin{aligned}
\sum_{i=1}^{n-1} \lfloor \lg i \rfloor &= \sum_{i=1}^{\lfloor \lg n \rfloor - 1} i \cdot 2^i + \left(n - 2^{\lfloor \lg n \rfloor}\right) \lfloor \lg n \rfloor \\
&= 2^{\lfloor \lg n \rfloor}(\lfloor \lg n \rfloor - 2) + 2 + \left(n - 2^{\lfloor \lg n \rfloor}\right) \lfloor \lg n \rfloor \\
&= n \lfloor \lg n \rfloor - 2^{\lfloor \lg n \rfloor + 1} + 2
\end{aligned}$$

Summary of Internal Sorting

☞ No one method is best for all conditions.

- Insertion sort is good when the list is already partially ordered. And it is best for small number of n.
- Merge sort has the best worst-case behavior but needs more storage than heap sort.
- Quick sort has the best average behavior, but its worst-case behavior is $O(n^2)$.
- The behavior of radix sort depends on the size of the keys and the choice of r.

Comparison

Algorithm	Time Complexity	Notes
insertion sort	$O(n^2)$	slow stable for small data sets ($<$ 1K)
quick sort	$O(n \log n)$ expected	unstable fastest (good for large inputs)
merge sort	$O(n \log n)$	stable sequential data access for huge data sets ($>$ 1M)
heap sort	$O(n \log n)$	fast unstable for large data sets (1K - 1M)

Radix Sort

Sorting Several Keys

- Consider the problem sorting records on several keys, $K^1, K^2, \ldots, K^r$ (K^1 is the most significant key). A list of records $R_1, R_2, \ldots, R_n$ are said to be sorted with respect to the keys $K^1, K^2, \ldots, K^r$ iff for every pair of records i and j, $i < j$ and $(K_i^1, K_i^2, \ldots, K_i^r) \leq (K_j^1, K_j^2, \ldots, K_j^r)$.
- $(x_1, x_2, \ldots, x_r) \leq (y_1, y_2, \ldots, y_r)$ iff either $x_i = y_i$, $1 \leq i \leq j$, and $x_{j+1} < y_{j+1}$ for some $j < r$ or $x_i = y_i, 1 \leq i \leq r$.
- Two popular ways to sort on multiple keys (K^1: suits, K^2: values)
  - **Most-significant-digit-first (MSD):**
    Ex: 2♣, . . . , A♣, 2♦, . . . , A♦, 2♥, . . . , A♥, 2♠, . . . , A♠
    1: sort on suits ➠ four piles
    2: sort on face values for each pile independently.
  - **Least significant digit first (LSD)**
    1: sort on face values
    ➠ 2♠, 2♣, 2♥, 2♦, 3♥, 3♠, 3♣, 3♦, . . . , A♣, A♥, A♠, A♦
    2: sort on suit for the whole list (stable sort)

MSD and LSD

- LSD and MSD only defines the order in which the keys are to be sorted. But they do not specify how each key is sorted.
- Bin sort can be used for sorting on each key. The complexity of bin sort is $O(n)$ if there are n bins.
- LSD and MSD can be used even when there is only one key.
- If the keys are numeric, then each decimal digit may be regarded as a subkey.
  ➠ Radix sort.
- In radix sort, we decompose the sort key using some **radix** r.
  - The number of bins needed is r.
  - Each key has d digits in the range of 0 to $r - 1$.

Radix Sort

```
int radixSort(element a[], int link[], int d, int r, int n)
{
     int front[r], rear[r];
     int i, bin, current, first, last;
     first = 1;
     for( i = 1; i < n; i++) link[i] = i+1;
     link[n] = 0;
     for( i = d-1; i>=0; i--) {
          for(bin = 0; bin < r; bin++) front[bin] = 0;
          for(current = first; current; current = link[current]) {
               bin = digit(a[current], i, r);
               if( front[bin] == 0) front[bin] = current;
               else link[rear[bin]] = current;
               rear[bin] = current;
          }
          for(bin = 0; !front[bin]; bin++);
          first = front[bin];
          last = rear[bin];
          for(bin++; bin < r; bin++)
               if(front[bin]) {
                    link[last] =front[bin];
                    last = rear[bin];
               }
          link[last] = 0;
     }
     return first;
}
```

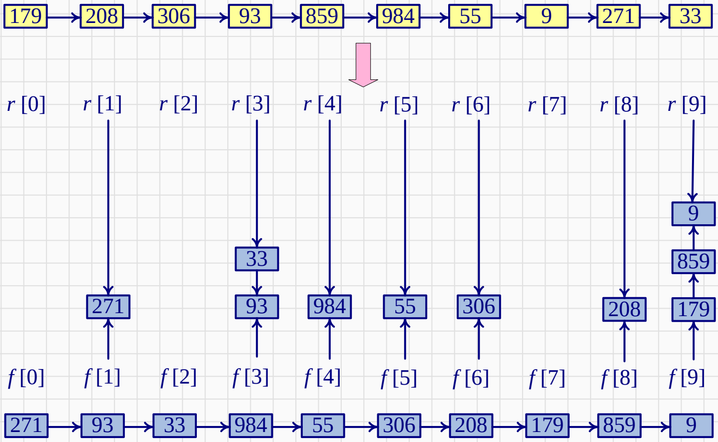
$O(n)$

$O(r)$

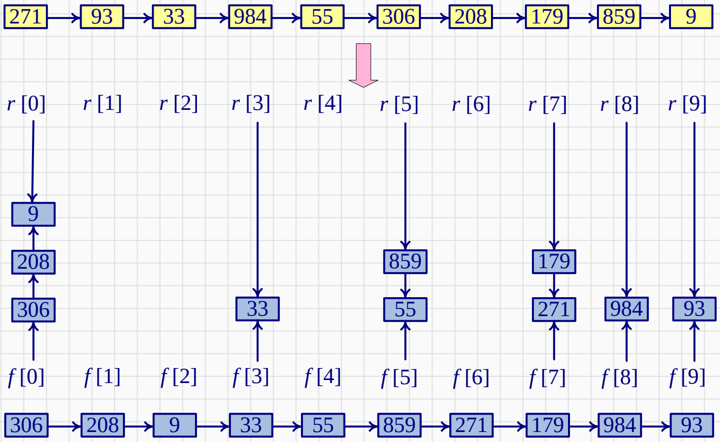
d

$\boxed{O(d(n+r))}$

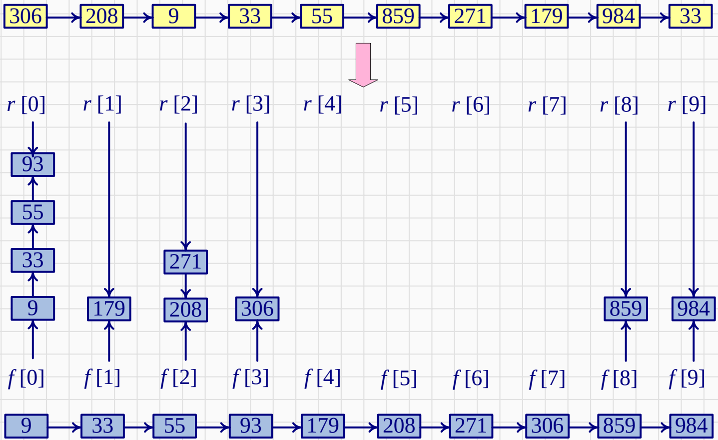
Example: Radix Sort (1/3)



Example: Radix Sort (2/3)



Example: Radix Sort (3/3)



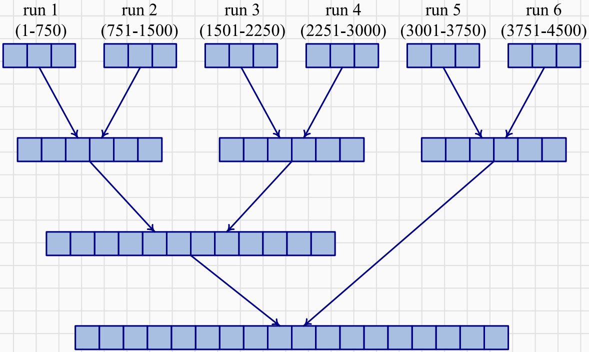
External Sorting

External Sorting

- There are some lists that are too large to fit in the memory of a computer. So internal sorting is not possible.
- Some records are stored in the disk (tape, etc.). System retrieves a block of data from a disk at a time. A block contains multiple records.
- The most popular method for sorting on external storage devices is merge sort.
  - Segments of the input list are sorted.
  - Sorted segments (called runs) are written onto external storage.
  - Runs are merged until only a run is left.
- Three factors contributing to the read/write time of disk:
  - seek time
  - latency time
  - transmission time

Example: External Sort (1/2)

- Consider a computer which is capable of sorting 750 records is used to sort 4500 records.
- Six runs are generated with each run sorting 750 records $\Longrightarrow$ **4 passes**



Example: External Sort (2/2)

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 $= t_s + t_\ell + t_{\text{rw}}$
t_{IS} = time to internally sort **750** records
nt_m = time to merge n records from input buffers to the output buffer

Analysis:

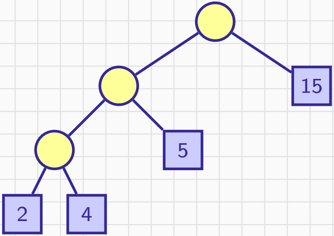
	Operation	Time
1	read 18 blocks of input, $18t_{\text{IO}}$, internally sort, $6t_{\text{IS}}$, write 18 blocks, $18t_{\text{IO}}$	$36t_{\text{IO}} + 6t_{\text{IS}}$
2	merge runs 1 to 6 in pairs	$36t_{\text{IO}} + 4500t_m$
3	merge two runs of 1500 records each, 12 blocks	$24t_{\text{IO}} + 3000t_m$
4	merge one run of 3000 records with one run of 1500 records	$36t_{\text{IO}} + 4500t_m$

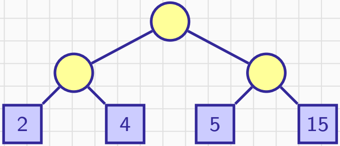
$132t_{\text{IO}} + 12000t_m + 6t_{\text{IS}}$

k-Way Merging

- To merge m runs via 2-way merging will need $\lceil \log_2 m \rceil + 1$ passes.
- If we use higher order merge, the number of passes over would be reduced.
- With k-way merge on m runs, we need $\lceil \log_k m \rceil + 1$ passes over.
- But is it always true that the higher order of merging, the less computing time we will have?
  - Not necessary!
  - $k - 1$ comparisons are needed to determine the next output.
  - If loser tree is used to reduce the number of comparisons, we can achieve complexity of $O(n \log_2 m)$
  - The data block size reduced as k increases. Reduced block size implies the increase of data passes over

Optimal Merging of Runs





weighted external path length

$= 2 \cdot 3 + 4 \cdot 3 + 5 \cdot 2 + 15 \cdot 1$

$= 43$

weighted external path length

$= 2 \cdot 2 + 4 \cdot 2 + 5 \cdot 2 + 15 \cdot 2$

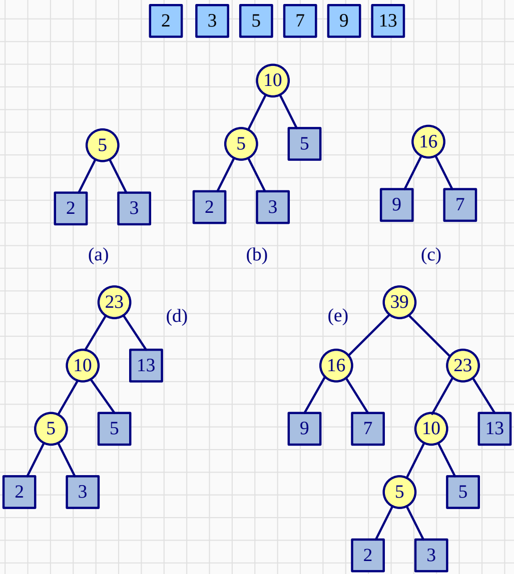
$= 52$

Huffman Algorithm

```
void huffman(treePointer heap[], int n)
{/* heap[1:n] */
    treePointer tree;
    int i;
    initialize(heap, n);

    for (i = 1; i < n; i++) {
        MALLOC(tree, sizeof(*tree));
        tree->leftChild = pop(&n);
        tree->rightChild = pop(&n);
        tree->weight = tree->leftChild->weight + tree->rightChild->weight;
        push(tree,&n);
    }
}
```

Example: Huffman Tree



Application: File Compression

☞ 100 characters ➜ $\lceil \log_2 100 \rceil = 7$ bits

- Using codes for characters in a file to reduce the file size
- Example:
  File contains only characters a, e, i, s, t, blank spaces, and newlines
  - Use three bits to code each character
  - size $= 3 \cdot 58 = 174$

Character	Code	Frequency	Total Bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
space	101	13	39
newline	110	1	3
Total		58	174

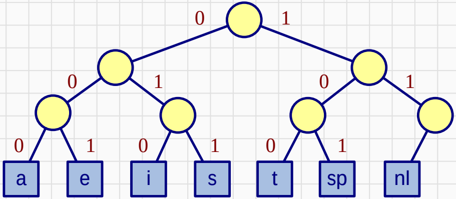
- Question: Can we use other coding to reduce the file size?
  ➠ Yes, by allowing code length to vary from character to character
- Short codes for frequently occurring characters

Tree Representation of Code

Cost: $\sum_i d_i \; f_i$

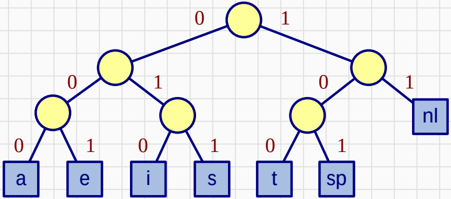
d_i : depth of code

f_i : frequency of code



code length of “nl” can be reduced by moving it to parent node

a better coding



File Compression Problem

- Property of tree representation of optimal code
  ➠ all nodes either are leaves or have two children
- Prefix code
  ➠ No character code is a prefix of another character code
  - No character is contained in non-leaf node
  - Provide unambiguous decoding
- Problem of file compression
  Find a full binary tree with minimum total cost, where all characters are in the leaves.

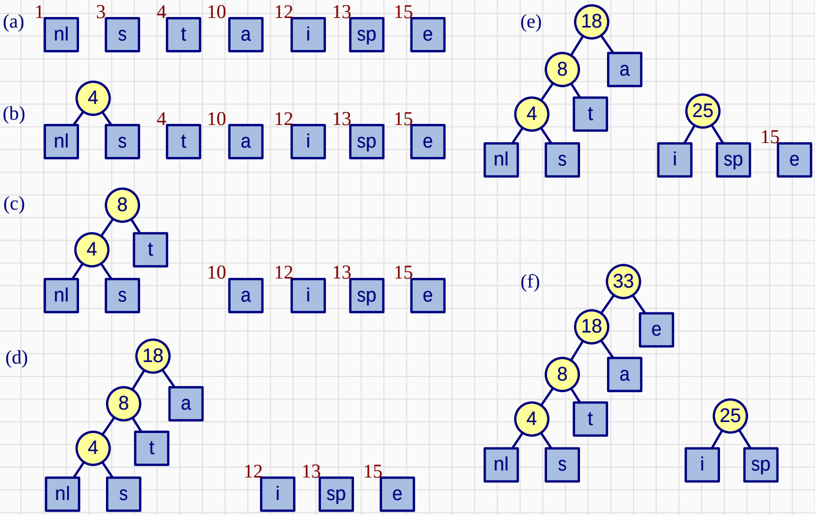
Huffman's Codes

Huffman's algorithm for file compression

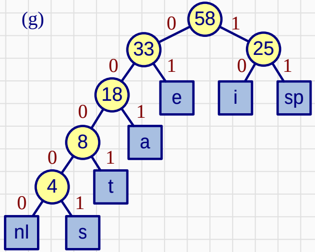
- A greedy method
- Outline of the algorithm:
  - Maintain a forest of trees: Initially, each character is a tree
  - Weight of tree = sum of frequencies of its leaves
  - Select two trees T_1 and T_2 with smallest weights to form a new tree with subtrees T_1 and T_2 until one tree remains

Character	Frequency
a	10
e	15
i	12
s	3
t	4
space	13
newline	1

Steps of Huffman's Algorithm



Huffman Code Assignment



Character	Code	Frequency	Total Bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00001	3	15
t	0001	4	16
space	11	13	26
newline	00000	1	5
Total		58	146

100010000101 ➠ iase