EECS 204002
Data Structures 資料結構
Prof. REN-SONG TSAY 蔡仁松 教授
NTHU

CH. 7 SORTING

7.1 Motivation

7.1

Motivation: Example

- Given a collection of records (***list***), where each record contains one or more fields (***keys***), ***how do we search a record with specific key?***
- Example

List	Phone book
Record	Person
Key	Name, Phone, Address, etc.
Searching	Find Jack.

Motivation: Sequential Search

- Search the WHOLE list in left-to-right or right-to-left order until we find the first occurrence of the record with the target key.

```
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;
}
```

Time complexity = $O(n)$

4

Motivation: Improvement

- How do we improve the performance of searching a record?
- Sort the list in a specific order before you do the search!
- For examples, given an ordered numeric list, using Binary search could obtain an improved performance of $O(\log n)$

5

Recursive Binary Search

```
int BinarySearch(int *A, const int x, const int
                   left, const int right )
{ // Search the A[left],..,A[right] for x
  if (left <= right) { // more integers to check
    int middle = (left+right)/2;
    if (x < A[middle])
       return BinarySearch(A, x, left, middle-1);
    else if (x > A[middle])
       return BinarySearch(A, x, middle+1, right);
    return middle;
  } // end of if
  return -1; // not found
}
```

6

Binary Search Example

- Search for $x = 9$ in array A[0]…[7] :

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
A	1	3	5	8	9	17	32	50
				1st	3rd	2nd		

1st call: `BinarySearch(A, 9, 0, 7)`

2nd call: `BinarySearch(A, 9, 4, 7)`

3rd call: `BinarySearch(A, 9, 4, 4)`

return index 4.

Why Need Sorting?

To improve the search performance!

Two Categories

- ***Internal*** sort:
  - The entire sort could be done in main memory
  - Suitable for list of small size (e.g. 1MB)
  - Insertion sort, merge sort, heap sort, radix sort
- ***External*** sort:
  - Data I/O are necessary during the sorting.
  - Suitable for list of large size (e.g. 1T)
  - Merge sort

Stable Sort

- A sort algorithm is called "***Stable***" iff $r_i = r_j$ and r_i precedes r_j in the input list, then r_i precedes r_j in the sorted list

Unsorted		Stable sort
21, 4, 5, 78, 5, 12	→	4, 5, 5 12, 21, 78

Unsorted		Unstable sort
21, 4, 5, 78, 5, 12	→	4, 5, 5 12, 21, 78

10

7.2 Insertion Sort

2018/12/4 © Ren-Song Tsay, NTHU, Taiwan 11

7.2

Insertion Sort

- Given a sequence $\boldsymbol{a}[1], \boldsymbol{a}[2], \ldots \boldsymbol{a}[\boldsymbol{n}]$
- Divide the sequence into 2 parts:
  - Left part: sequence sorted so far
  - Right part: unsorted part
- Take one element from the right part and **insert** it into the correct position in the left part

12

A Running Example

44 55 12 42 94 18 6 67

44 55 12 42 94 18 6 67

44 55 12 42 94 18 6 67

12 44 55 42 94 18 6 67

12 42 44 55 94 18 6 67

…

13

Insertion Sort (codes)

```
template <class T>
void Insert(cones T& e, T *a, int i){
      a[0] = e;
      while (e < a[i]) {
        a[i+1] = a[i];
        i--; }
      a[i + 1] = e;
}
template <class T>
void InsertionSort(T *a, const int n){
      for (int j = 2; j <= n ; j++){
        T temp = a[j];
        Insert(temp, a, j - 1);}
}
```

14

Complexity

- Worst case running time
  - Outer loop: $O(n)$
  - Inner loop: $O(j)$

$$\sum_{j=1}^{n} j = O(n^2)$$

- Average case running time: $O(n^2)$
- Stable sort

15

7.3

Quick Sort

7.3

Quick Sort

- Pick a record $\boldsymbol{a}[r]$ at random.
- Divide $a[1]$ … … $a[n]$ into two sublists using $a[r]$.

$a[i] \leq a[r]$	$a[r]$	$a[j] > a[r]$

- Sublists are not sorted.
- Sort the two sublists recursively.
- How to pick up a splitting record ?
  - Just pick up the first record!

Example

26	5	37	1	61	11	59	15	48	19
26	5	37	1	61	11	59	15	48	19
	i				swap				j
26	5	19	1	61	11	59	15	48	37
			i		swap			j	
26	5	19	1	15	11	59	61	48	37
		$i > j \rightarrow$ stop			j	i			
11	5	19	1	15	26	59	61	48	37
		Sublist 1					Sublist 2		

- Recursively sort sublist1 and sublist2

Quick Sort (code)

```
template <class T>
void QuickSort(T *a, const int left, const int right)
{       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);
        }
}
```

19

Time complexity

- If the splitting record is in the middle
- Depth of recursion: $O(\log n)$
- Finding the position of splitting record: $O(n)$
- Total running time: $O(n \log n)$
- Worst case running time: $O(n^2)$

$A[r]$ $n-1$

$n-2$

Ex: 1,2,3,4,5,6,7 a sorted list

20

Variation: Median-of-Three

- Find a better splitting record:
  - Try to find the median one
  - Median {first, middle, last}
- Not a stable sort.

21

7.4

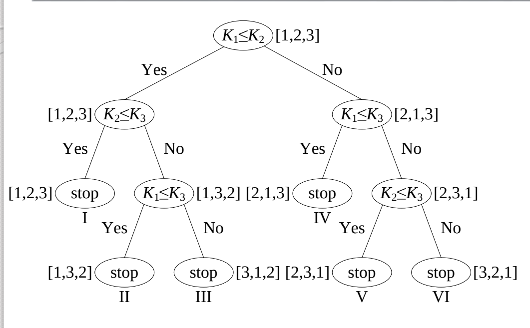
How Fast Can We Sort

7.4

Best Sorting Computing Time

- $\Omega(n \log n)$:
  - If only the comparisons and interchanges are allowed during the sorting
- Decision tree:
  - A tree that describe sorting process.
  - Each vertex represents a comparison.
  - Each branch indicates the result.

Decision Tree for Insertion Sort



Time Complexity

- Given a list of n records.
- There are $n!$ combinations and thus having $n!$ leaf nodes in a decision tree.
- For a decision tree (binary tree) with $n!$ leaves, the height (depth) of the tree is $n \log n$.
  - $n! \geq (n/2)^{n/2}$
  - $\Rightarrow \log(n!) \geq (n/2)\log(n/2) = \Omega(n \log n)$
- Therefore the average root-to-leaf path is $\Omega(n \log n)$.

25

7.5

Merge Sort

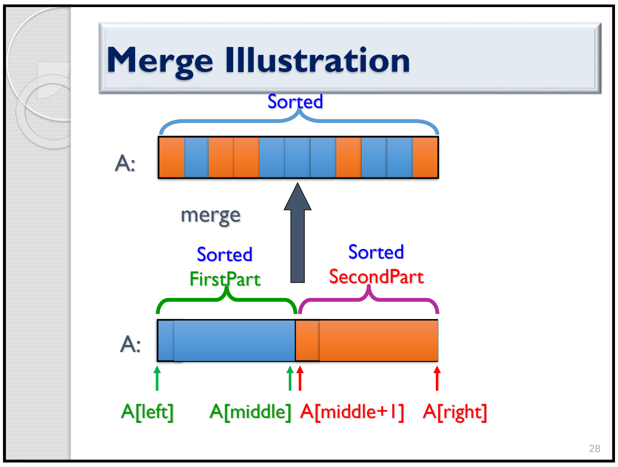
2018/12/4 © Ren-Song Tsay, NTHU, Taiwan 26

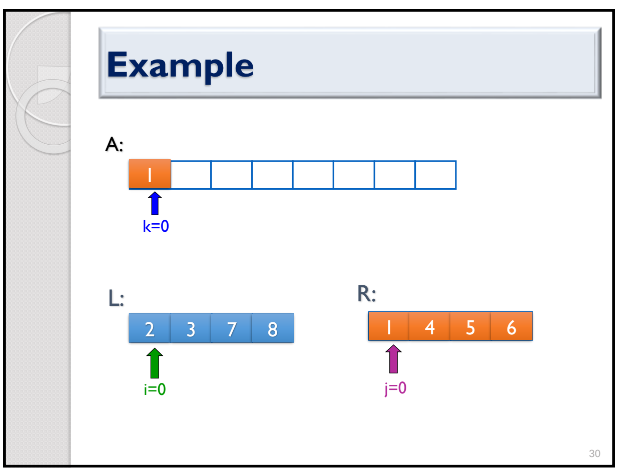
7.5

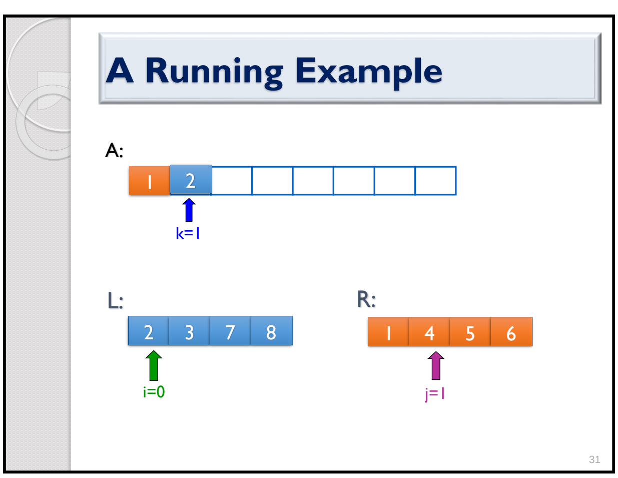
Merge Sort

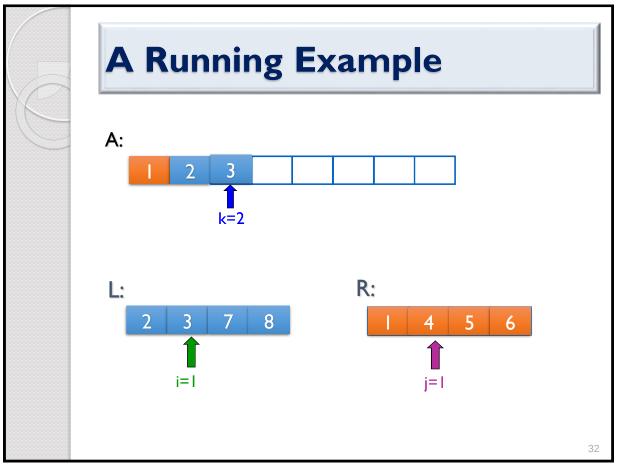
- Given two sorted lists, merge them into one sorted list.
- Use an algorithm similar to polynomial addition.
- Assume the size of two lists are m and l, the time complexity of merging two lists is $O(m + l)$.

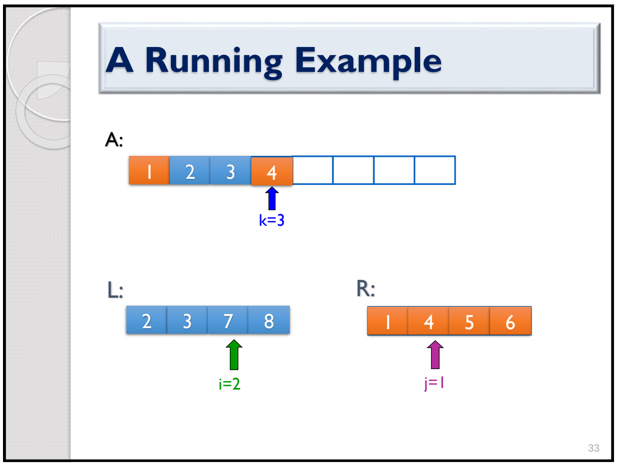
27

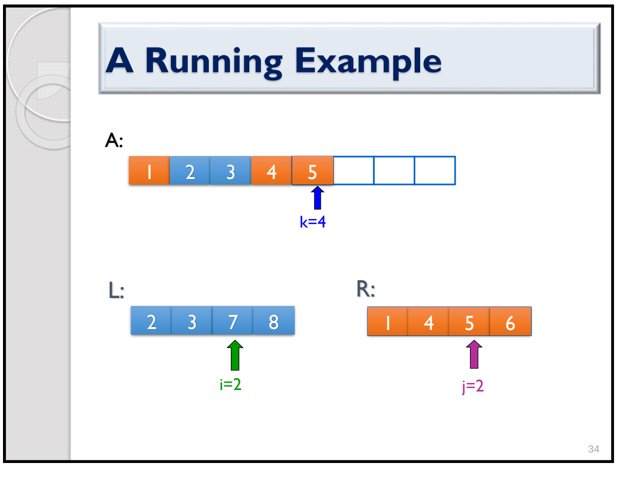

Merge Illustration
Sorted
A:
merge
Sorted
FirstPart
Sorted
SecondPart
A:
A[left]
A[middle]
A[middle+1]
A[right]
28

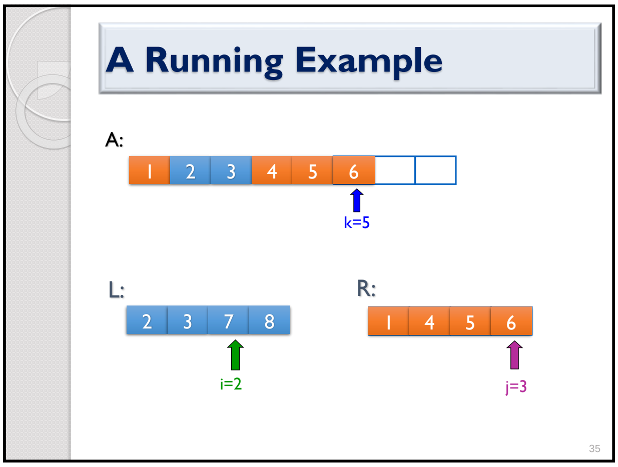

Example
A:
1
k=0
L:
2 3 7 8
i=0
R:
1 4 5 6
j=0
30

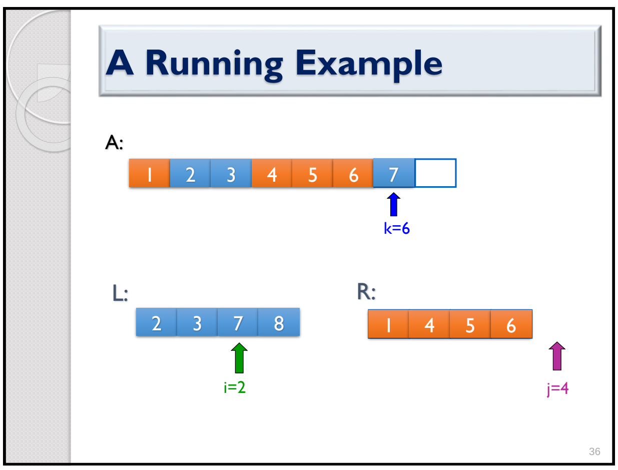

A Running Example
A:
1 2
k=1
L:
2 3 7 8
i=0
R:
1 4 5 6
j=1
31

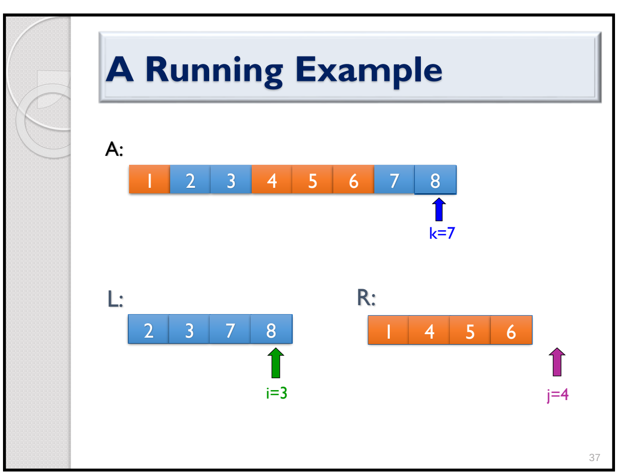

A Running Example
A:
1 2 3
k=2
L:
2 3 7 8
i=1
R:
1 4 5 6
j=1
32


A Running Example
A:
1 2 3 4
k=3
L:
2 3 7 8
i=2
R:
1 4 5 6
j=1
33

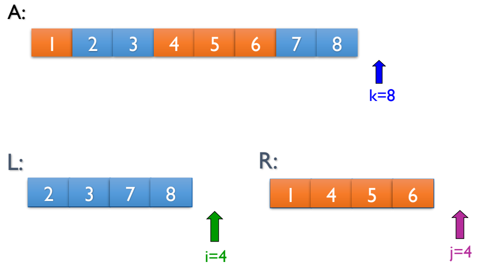

A Running Example
A:
1 2 3 4 5
k=4
L:
2 3 7 8
i=2
R:
1 4 5 6
j=2
34


A Running Example
A:
1 2 3 4 5 6
k=5
L:
2 3 7 8
i=2
R:
1 4 5 6
j=3
35


A Running Example
A:
1 2 3 4 5 6 7
k=6
L:
2 3 7 8
i=2
R:
1 4 5 6
j=4
36


A Running Example
A:
1 2 3 4 5 6 7 8
k=7
L:
2 3 7 8
i=3
R:
1 4 5 6
j=4
37

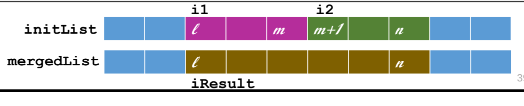
A Running Example



38

Merge Sort (codes)

```
template <class T>
void Merge(T *initList, T *mergedList, const int ℓ, const int m,
const int n)
{ for (int i1 = ℓ, iResult = ℓ, i2 = m + 1; i1 <= m && i2 <= n;
      iResult++)
      if (initList[i1] <= initList [i2]){
            mergedList[iResult] = initList[i1];
            i1++;
      }else{
            mergedList[iResult] = initList[i2];
            i2++;}
  // copy the remaining records, if any, of the 1st list
  copy (initList + i1, initList + m + 1, mergedList + iResult);
  // copy the remaining records, if any, of 2nd list
  copy (initList + i2, initList + n + 1, mergedList + iResult);
}
```



39

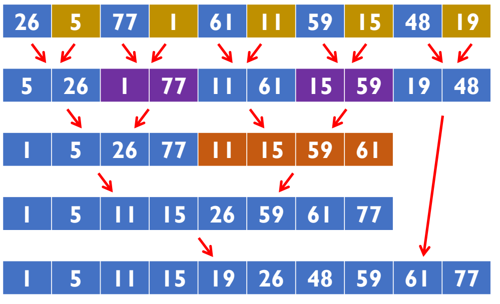
7.5.2

Iterative Merge Sort

- Interpret the list as comprised of **n sorted sublists.**
- 1^{st} merge pass: n **sublists** are merged by pairs to obtain $n/2$ **sublists**.
- 2^{nd} merge pass: $n/2$ **sublists** are merged by pairs to obtain $n/4$ **sublists.**
- …
- The process repeats until only one sublist exists.

40

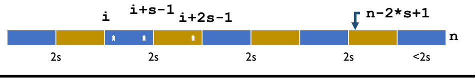
Example



41

Iterative Merge Sort (code)

```
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 size of initList.
 for (int i = 1; // i is the 1st position in the 1st sublist
      i <= n-2*s+1; // enough records for two sublists?
      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);
}
```



42

Iterative Merge Sort (code)

```
template <class T>
void MergeSort(T *a, const int n)
{
  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); // switch role of a and
                                    // tempList
  }
  delete [] tempList;
}
```

43

Properties

- Time complexity
  - Number of merge pass: $O(\log n)$
  - Time complexity of merge pass: $O(n)$
  - Time complexity = $O(n \log n)$
- Require additional storage to store merged result during the process.
- Stable sort

44

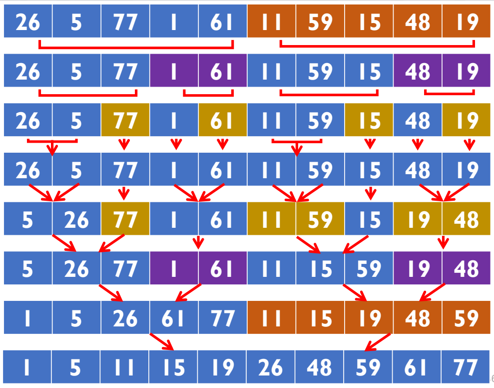
7.5.3

Recursive Merge Sort

- Divide the list to be sorted into two roughly equal parts called **left and right sublists**.
- Recursively sort the two sublists.
- Merge the sorted sublists

45

Recursive Merge Sort Example



Recursive Merge Sort (code)

- Using a structure "link" to represent the index order of sorted list.

```
template <class T>
int rMergeSort(T* a, int* link, const int left, const int right)
{// sorting a[left:right]. link[i] is initialize to 0.
 // rMerge returns the index of 1st element in the sorted list.
  if (left >= right) return left;
  int mid = (left + right) /2;
  return ListMerge(a, link,
      rMergeSort(a, link, left, mid),      // sort left sublist.
      rMergeSort(a, link, mid + 1, right));// sort right sublist.
}
```

47

```
tamplate <class T>
int ListMerge(T* a, int* link, const int start1, const int
start2)
{// merge two sorted lists, starting from start1 and start2.
 // link[0] is a temporary head, stores the head of merged list.
 // iRsults records the last element of currently merged list.
 int iResult = 0;
 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 the remaining list to the resultant list.
 if (i1 = = 0) link[iResult] = i2;
 else link[iResult] = i1;
 return link[0];
}
```

		1	2	3	4	5	6	7	8	9	10
index		1	2	3	4	5	6	7	8	9	10
data		26	5	77	1	61	11	59	15	48	19
link	4	9	6	0	2	3	8	5	10	7	1

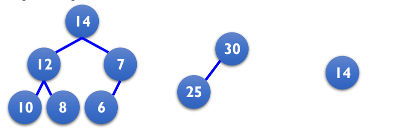
7.6

Heap Sort

2018/12/4 © Ren-Song Tsay, NTHU, Taiwan 49

Max Heap (Priority Queue)

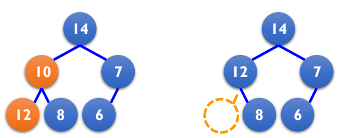
Definition: A ***max (min) tree*** is a tree in which the key value in each node is ***no smaller*** (***larger***) than the key values in its children (if any). A ***max(min) heap*** is a ***complete binary tree*** that is also a ***max(min) tree***.



Max Heap Max Heap Max/Min Heap

50

Examples: not max heap



Not a heap (12 > 10)

Not a heap (Not a complete binary tree)

51

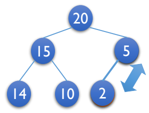
Max Heap: Representation

- Since the heap is a complete binary tree, we could adopt "**Array Representation**" as we mentioned before!
- Let node i be in position i (array[0] is empty)
  - $\boldsymbol{Parent(i) = \lfloor i/2 \rfloor}$ if $i \neq 1$. If $i = 1$, i is the root and has no parent.
  - $\boldsymbol{leftChild(i) = 2i}$ if $2i \leq n$. If $2i > n$, then i has no left child.
  - $\boldsymbol{rightChild(i) = 2i + 1}$ if $2i + 1 \leq n$, if $2i + 1 > n$, then i has no right child.

52

Max Heap: Insert

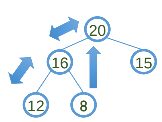
- Make sure it is a complete binary tree
- Insert a new node
- Check if the new node is greater than its parent
- If so, swap two nodes



53

Max Heap: Delete

1. Always delete the root
2. Move the last element to the root (maintain a complete binary tree)
3. Swap with larger and largest child (if any)
4. Continue step 3 until the max heap is maintained (trickle down)



54

7.6

Heap Sort

- Utilize the max-heap structure
- The insertion and deletion could be done in O(logn)
- Build a max-heap using n records, insert each record one by one (O(nlogn))
- Iteratively remove the largest record (the root) from the max-heap (O(nlogn))
- Not a stable sort

2018/12/4 © Ren-Song Tsay, NTHU, Taiwan 55

Heap Sort (code)

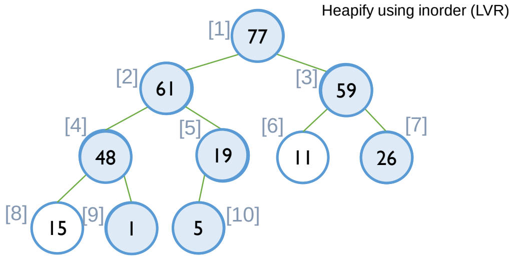
```
template <class T>
void HeapSort(T *a, const int n)
{
  Heapify(a, n);
  for (i = n-1; i >= 1; i--)  // Sorting
  {
      swap(a[1], a[i+1]);     // swap the root with last node
      Heapify(a, i);          // rebuild the heap (a[1:i])
  }
}
```

56

Heap Sort Example

26 5 77 1 61 11 59 15 48 19

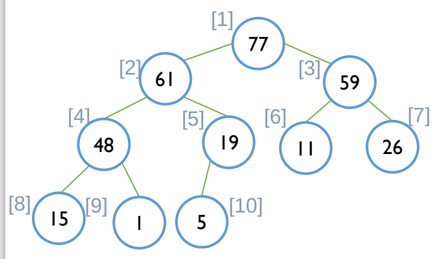
Heapify using inorder (LVR)



Yi-Shin Chen -- Data Structures 57

Heap Sort Example

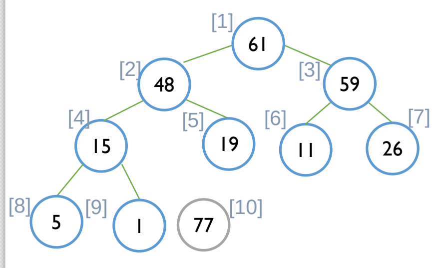
77 61 59 43 19 11 26 15 1 5



Yi-Shin Chen -- Data Structures 58

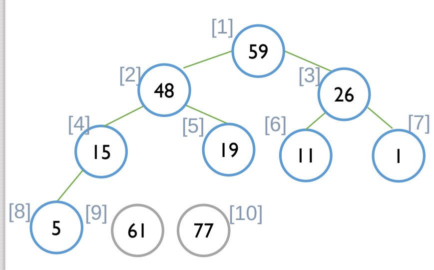
Heap Sort Example

61 48 59 15 19 11 26 5 1 | 77



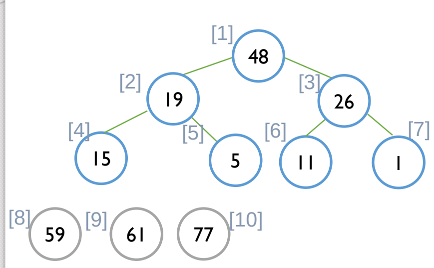
Heap Sort Example

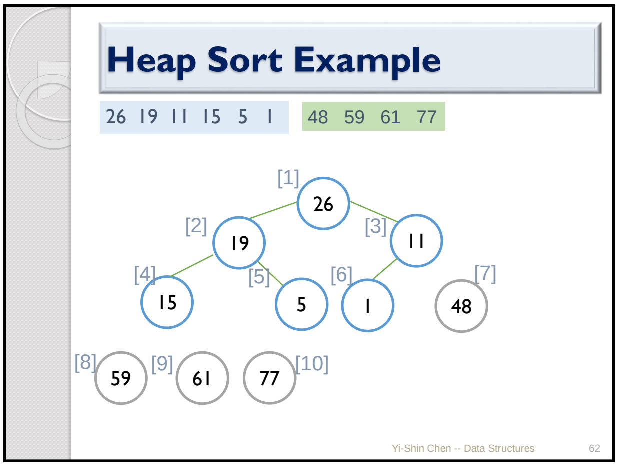
59 48 26 15 19 11 1 5 | 61 77

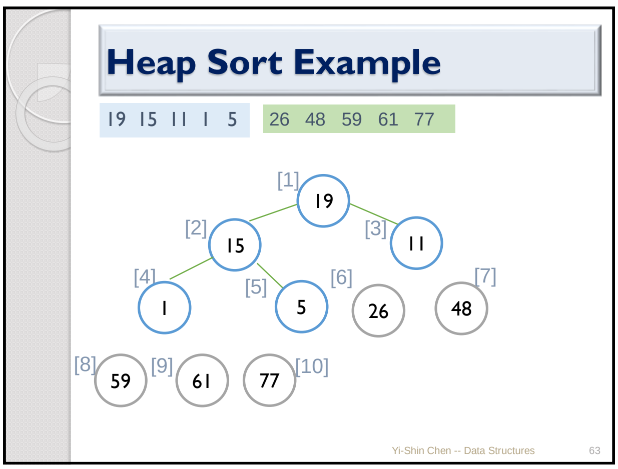


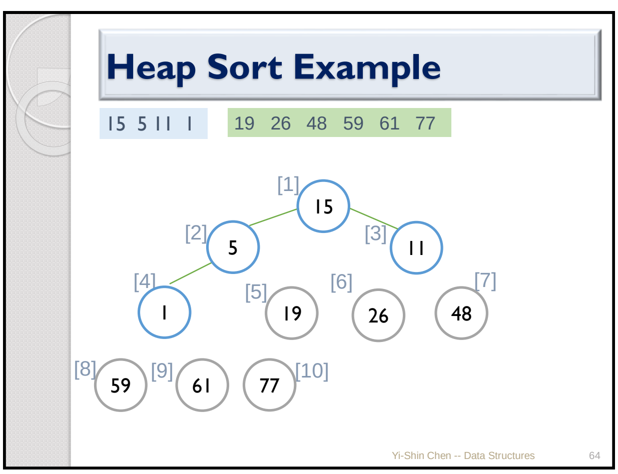
Heap Sort Example

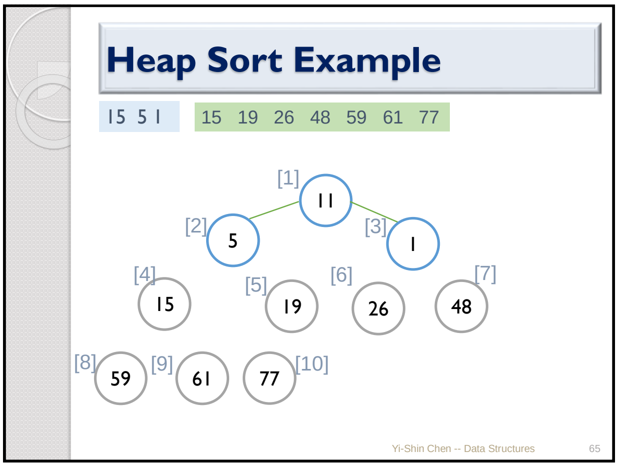
48 19 26 15 5 11 1 | 59 61 77

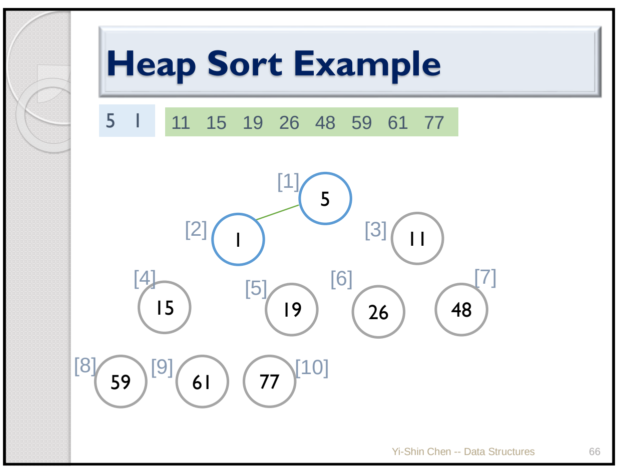


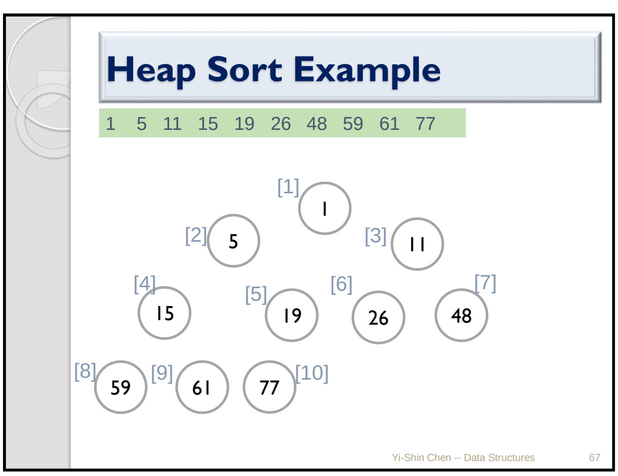

Heap Sort Example
26 19 11 15 5 1
48 59 61 77
[1] 26
[2] 19
[3] 11
[4] 15
[5] 5
[6] 1
[7] 48
[8] 59
[9] 61
[10] 77
Yi-Shin Chen -- Data Structures
62


Heap Sort Example
19 15 11 1 5
26 48 59 61 77
[1] 19
[2] 15
[3] 11
[4] 1
[5] 5
[6] 26
[7] 48
[8] 59
[9] 61
[10] 77
Yi-Shin Chen -- Data Structures
63


Heap Sort Example
15 5 11 1
19 26 48 59 61 77
[1] 15
[2] 5
[3] 11
[4] 1
[5] 19
[6] 26
[7] 48
[8] 59
[9] 61
[10] 77
Yi-Shin Chen -- Data Structures
64


Heap Sort Example
15 5 1
15 19 26 48 59 61 77
[1] 11
[2] 5
[3] 1
[4] 15
[5] 19
[6] 26
[7] 48
[8] 59
[9] 61
[10] 77
Yi-Shin Chen -- Data Structures 65


Heap Sort Example
5 1
11 15 19 26 48 59 61 77
[1] 5
[2] 1
[3] 11
[4] 15
[5] 19
[6] 26
[7] 48
[8] 59
[9] 61
[10] 77
Yi-Shin Chen -- Data Structures 66


Heap Sort Example
1 5 11 15 19 26 48 59 61 77
[1] 1
[2] 5
[3] 11
[4] 15
[5] 19
[6] 26
[7] 48
[8] 59
[9] 61
[10] 77
Yi-Shin Chen -- Data Structures 67

7.7

Sorting on Several Keys

7.7

Sorting with Several Keys

A list of records is 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

$$(K_i^1, K_i^2, \ldots, K_i^r) \le (K_j^1, K_j^2, \ldots, K_j^r)$$

$(x_1, \ldots, x_r) \le (y_1, \ldots, y_r)$
iff either $x_k = y_k, 1 \le k \le n$, and $x_{n+1} < y_{n+1}$ for some $n < r$,
or $x_k = y_k, 1 \le k \le r$

Sorting a Deck of Cards

- Each card has two keys
  - K^1 (Suits): ♣ < ♦ < ♥ < ♠
  - K^2 (Face values): 2 < 3 < 4 …<J < Q < K < A
  - The sorted list is: 2 ♣, …,A♣, …, 2 ♠, …,A ♠
- Most-significant-digit (**MSD**) sort
  - Sort using K^1 to obtain 4 "piles" of records.
  - Sort each piles into sub-piles.
  - Merge piles by placing the piles on top of each other.

Sorting a Deck of Cards (cont'd)

- Least-significant-digit (**LSD**) sort
  - Sort using K^2 to obtain 13 "piles" of records.
  - Place 3's on top of 2's,…, Aces on top of kings.
    - 2 < 3 < 4 … J < Q < K < A
  - Using a stable sort with respect to K^1 and obtain 4 "piles".
  - Merge piles by placing the piles on top of each other.

71

Bin Sort (Bucket Sort)

- Assume the records in a list to be sorted come from a set of size $\boldsymbol{m}$, say $\{1,2,\ldots,m\}$.
- Create $\boldsymbol{m}$ buckets.
- Scan the sequence $a[1]$ … $a[n]$, and put $a[i]$ element into the $\boldsymbol{a[i]^{th}}$ bucket.
- Concatenate all buckets to get the sorted list.
- Suitable for a set with small $\boldsymbol{m}$.

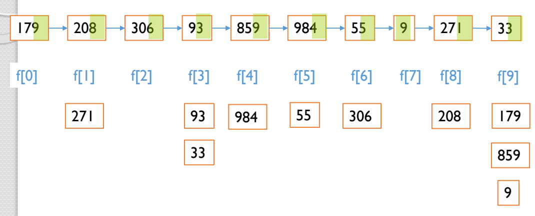
72

Radix Sort

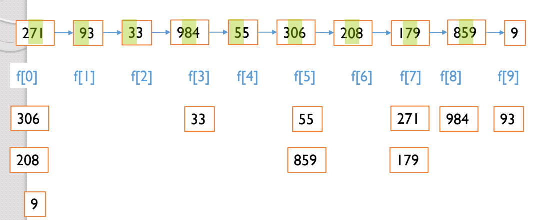
- Decompose the key (number) into subkeys using some **radix** $\boldsymbol{r}$
  - For $r = 10, K = 123$, then $K^1 = 1, K^2 = 2$, and $K^3 = 3$.
- Create $\boldsymbol{r}$ buckets (0 ~ $\boldsymbol{r-1}$).
- Apply bin sort with MSD or LSD order.
- Suitable to sort numbers with large value range.

73

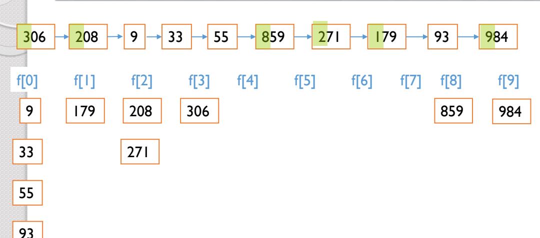
Radix Sort Example (Pass 1)



Radix Sort Example (Pass 2)



Radix Sort Example (Pass 3)



Time Complexity: O(d*(n+r))

LSB Radix Sort (code) 1/2

```
template <class T>
int RadixSort(T *a, int *link, const int d, const int r, const int n)
{// using a radix sort with d digits、radix r to sort a[1:n]
 // digit(a[i], j, r) return the j-th key in radix r of a[i]
 // each digit is within the range [0, r). Using the bin sort to
 // sort elements of the same digit.
 int e[r], f[r]; // head and tail of the bin
 int first = 1; // start from the 1st element
 for(int i =1; i < n; i++) link[i]=i+1; // link the elements
 link[n] = 0;
 // do radix sorting…
```

77

LSB Radix Sort (code) 2/2

```
  // do radix sorting…
  for (i = d-1; i >=0; i--) { // sort in LSB order
    fill(f, f+r, 0); // initialize the bins
    for (int current = first; current; current = link[current])
    { // put the element with key k to bin[k]
      int k = digit(a[current], i, r);
      if (f[k]== 0) f[k] = current;
      else link[e[k]] = current;
      e[k] =current;
    }
    for (j = 0; !f[j]; j++); // find the 1st non-empty bin
    first = f [j];
    int last = e[j];
    for (int k = j + 1; k < r; k++){ // link the rest of bins
      if (f[k]) {
         link[last] = f[k];
         last = e[k];}
    }
    link[last] = 0;
  }
  return first;
}
```

78

7.9

Summary of Internal Sorting

2018/12/4 © Ren-Song Tsay, NTHU, Taiwan 79

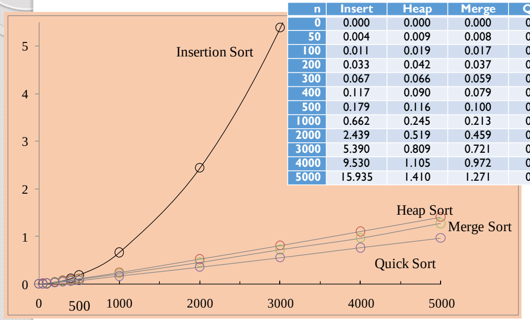
7.9

Time Complexity Comparison

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

80

Actual Runtime Comparison



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

81

Design Guidelines

- Insertion sort is good for small n and when the list is partially sorted.
- Merge sort is slightly faster than heap sort but it require additional storage.
- Quick sort outperforms in average.
- Combining insertion sort with quick sort to obtain better performance.

82

C++'s Sort Methods

- Designed to optimize the average performance.
- std::sort()
  - Modified Quick sort.
  - Heap Sort
    - when the number of subdivision exceed $c\log n$
  - Insertion Sort
    - when the segment size becomes small
- std::stable_sort()
  - Merge Sort.
  - Insertion Sort
    - when the segment size becomes small
- std::partial_sort()
  - Heap Sort.

83

7.10

External Sorting

2018/12/4 © Ren-Song Tsay, NTHU, Taiwan 84

7.10

External Sort

- When the lists are too large to be loaded into internal memory completely
  - The list could reside on a disk
- The external sorting operations
  - Read partial records
  - Perform the sorting
  - Write the result back to disk
- "Block"
  - The unit of data that is read/written at one time

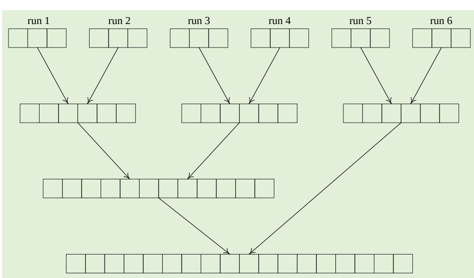
85

External Sort Algorithm

- Insertion sort, Quick sort, Heap sort…..NO
- **Merge sort**……………………………YES
  - Segments (blocks, runs) of input lists sorted using an internal sort
  - Sublists could be sorted independently and merged later
  - The runs generated in phase one are merged together following the **merge-tree** pattern
  - During the merging, only the leading records of the two runs needed to be loaded in memory

86

Runs & Merge Tree

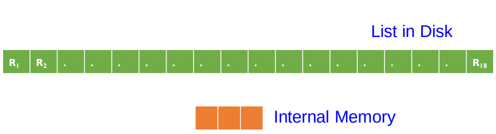


Merge tree

87

Example: Problem

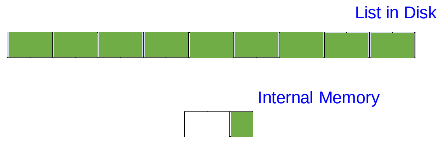
- Internal memory: 750 records.
- List to be sorted: 4500 records.
- Block size: 250 records.



88

Example: Merge Pass 1

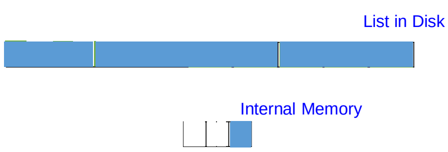
- To merge R_i and R_{i+1}:
  - The blocks of R_i and R_{i+1} are read into input buffers
  - The merged data is written to output buffer
  - Output buffer full ⇒ write onto disk
  - Input buffer empty ⇒ read from the new block



89

Example: Merge Pass 2

- To merge R_i and R_j:
  - The blocks of R_i and R_j are read into input buffers
  - The merged data is written to output buffer
  - Output buffer full ⇒ write onto disk
  - Input buffer empty ⇒ read from the new block

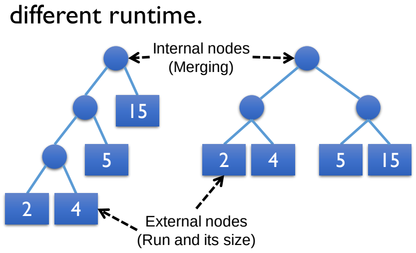


90

7.10.5

Optimal Merging of Runs

- Runs with different sizes.
- Different merge sequence may result in different runtime.

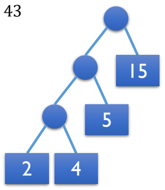


93

Runtime Evaluation

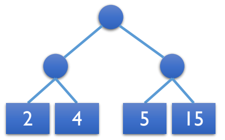
Merge tree A

$Cost$
$= (2+4)+(2+4+5)+(2+4+5+15)$
$= 2*3+4*3+5*2+15*1$
$= 43$



Merge tree B

$Cost$
$= 2*2+4*2+5*2+15*2 = 52$



94

Weighted External Path Length

- The total number of merge steps is equal to:

$$\sum_{i=1}^{n} s_i d_i$$

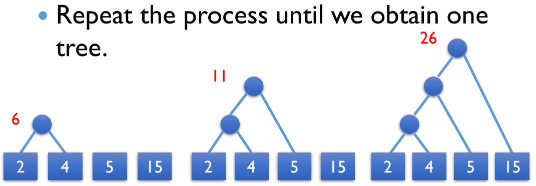
- Where s_i is the size of Run i and d_i is the distance from the node to root.
- **How to build a merge tree such that the total cost is minimized?**

95

Sort by Block Size

- Sort runs using its size.

- Take the two runs with **least sizes** and combine them into a tree.
- Repeat the process until we obtain one tree.



96

Similar to Message Encoding

- Given a set of messages $\{M_1, M_2, \dots, M_i\}$
- How do we encode each M_i using a binary code such that the total number of message bits is minimum?

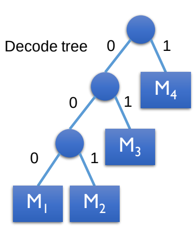
	Encode 1	Encode 2	Encode 3
M_1	0	0001	0001
M_2	1	0010	1
M_3	10	0100	01
M_4	11	1000	001

97

7.10.5
F7.28

Huffman Code

- Using a binary tree, called **decode tree** to encode messages.



	Huffman Code
M_1	000
M_2	001
M_3	01
M_4	1

99

Decoding Cost

- Cost of decoding a code word is proportional to the number of bits of the word.
  - Decoding a code word contain $2 * M_1$ and $1 * M_4$ requires process $2 * 3 + 1 = 7$ bits.
- Assume the message M_i with encoded bit length d_i, occurring frequency is s_i, then the total cost of the code word is:

$$\sum_{i=1}^{n} s_i d_i$$

- **How do we construct a decode tree such that the decoding cost is minimized?**

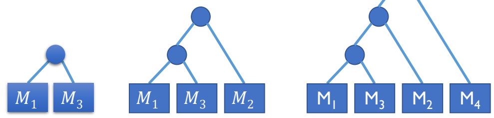
100

Optimal Merge Tree

- Follow Huffman Code Method
- Sort the message according to s_i

M_1	M_3	M_2	M_4
2	4	5	15

- Take two messages with the **least** $\boldsymbol{s_i}$ and combine them into a tree (a new message)
- Repeat the process until we obtain one tree.



101

Self-Study Topics

- 7.8 List and Table Sorts

103