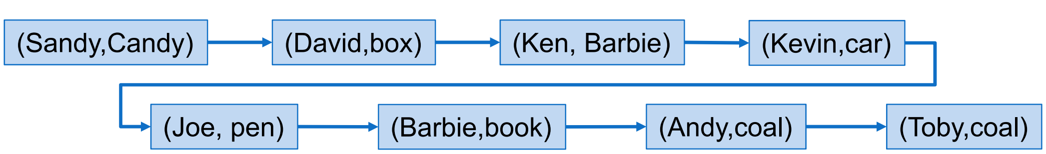
Sorting

Yi-Shin Chen
Institute of Information Systems and Applications
Department of Computer Science
National Tsing Hua University
yishin@gmail.com

Motivation

- Given a list, where each record contains one or more keys, how do we search a record with a specific key?
- Sequential search
  - Search the list in left-to-right or right-to-left order until we find the first occurrence of the record with the key
  - Complexity: O(N)



Improvement?

- Sort the list in a specific order before searching

- Approaches
  - Insertion based on some sorting policy
    - Retrieval time should be small
  - Sort after a batch of insertion
    - Insertion time should be small
    - Chance of retrieval is rare

Categories of Sorting

- Internal sort
  - The entire sort could be done in main memory
  - Suitable for list of small size (e.g. 1MB)
  - Types: 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)
  - Types: Merge sort

Stable Sort

- Stable sort algorithms can keep
  - 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

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

Insertion Sort

Motivation of Insertion Sort

- Two parts in the input sequence
  - Sorted one: the left part
  - Unsorting one: the right part
- Sort one element one at a time
  - Take one from the right part and insert it into the correct position in the left part

Algorithm of Insertion Sort

```
template <class T>
void Insert(const 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);}
}
```

Properties

- Worst case running time
  - Outer loop: O(n)
  - Inner loop: O(j)
  - Total running time: $O(n^2)$
- Average case running time: $O(n^2)$
- Stable sort

Quick Sort

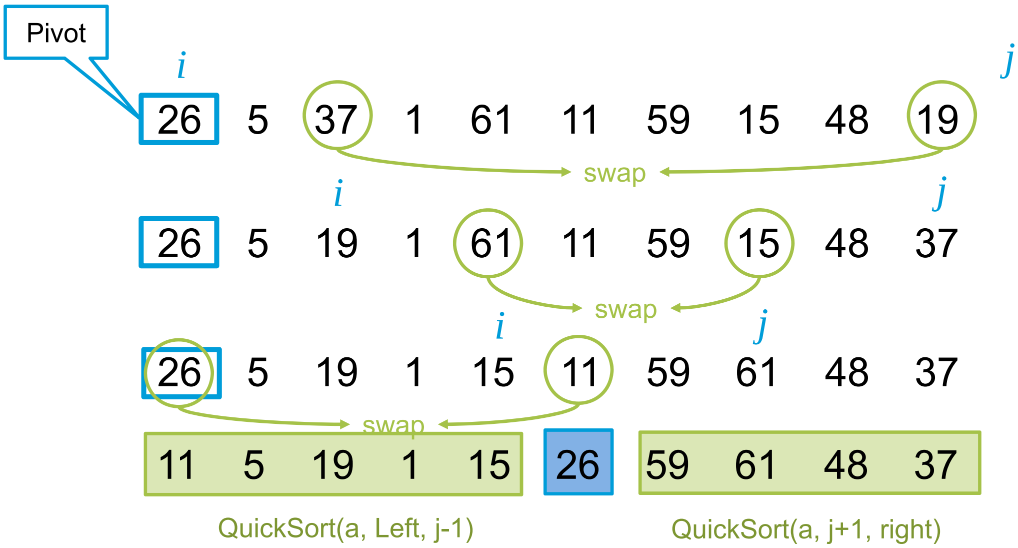
Motivation of Quick Sort

- Divide and conquer
- Utilize a "Pivot“
  - The left records of the pivot are less than or equal to that of the pivot
  - The right records of the pivot are greater than that of the pivot
- Steps
  - Find the position of the selected pivot
  - Sort the two sublists recursively

Quick Sort (Codes)

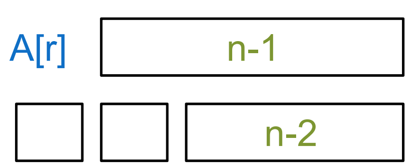
```
template <class T>
void QuickSort(T *a, const int left, const int right)
{       if (left < right) {
                int i = left, j = right + 1, pivot = a[left];
                do {
                        do i++; while (a[i] < pivot);
                        do j--; while (a[j] > pivot);
                        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);
        }
}
```

Quick Sort Example



Time Complexity

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



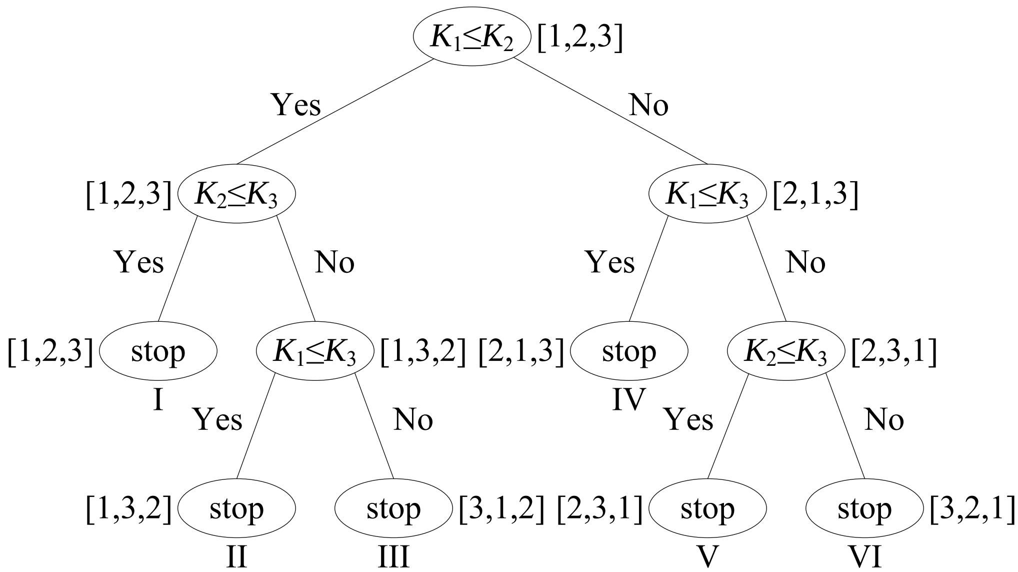
Properties

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

How Fast Can We Sort?

- What is the best computing time for sorting?
  - If only comparisons and interchanges during sorting
    - $\Omega(nlogn)$ is the best possible time
- Decision tree:
  - A tree that describe sorting process
  - Each vertex represents a comparison
  - Each branch indicate the result

Decision Tree for Insertion Sort



Time Complexity

- Given a list of *n* records
  - *n!* combinations and n! leaf nodes in a decision tree
  - The height (depth) of the tree is *nlogn*
- Therefore the average root-to-leaf path is ***Ω(nlogn)***

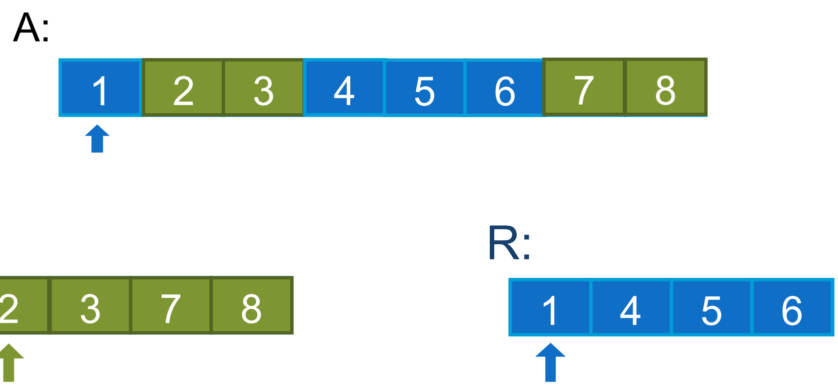
Merge Sort

Motivation of Merge Sort

- Merge sorted lists to get a single sorted one
- Divide and conquer
  - Divide till the lists are sorted
  - Merge lists recursively
- Stable sort

Merging

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



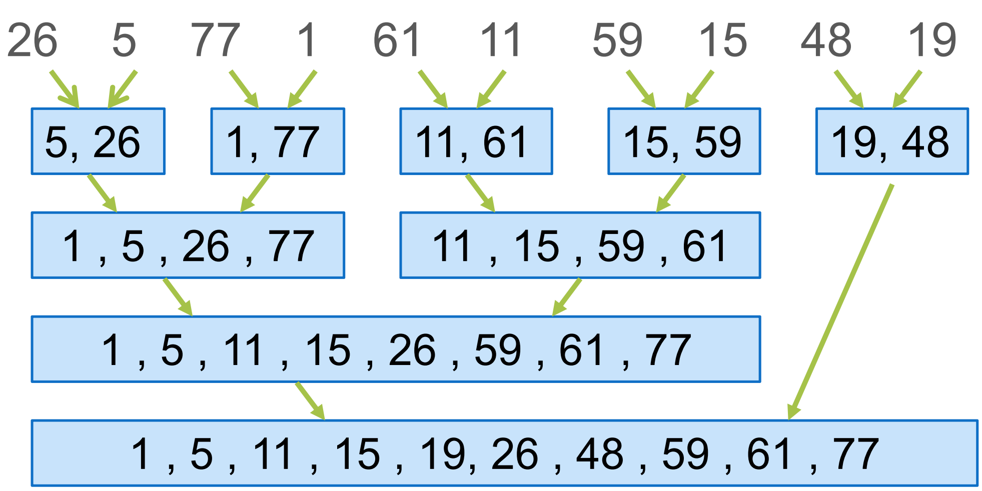
Merging (Code)

```
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 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);
}
```

Iterative Merge Sort

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

MergePass Example



Iterative Merge Sort (codes)

```
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);
}
```


Iterative Merge Sort (codes)

```
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;
}
```

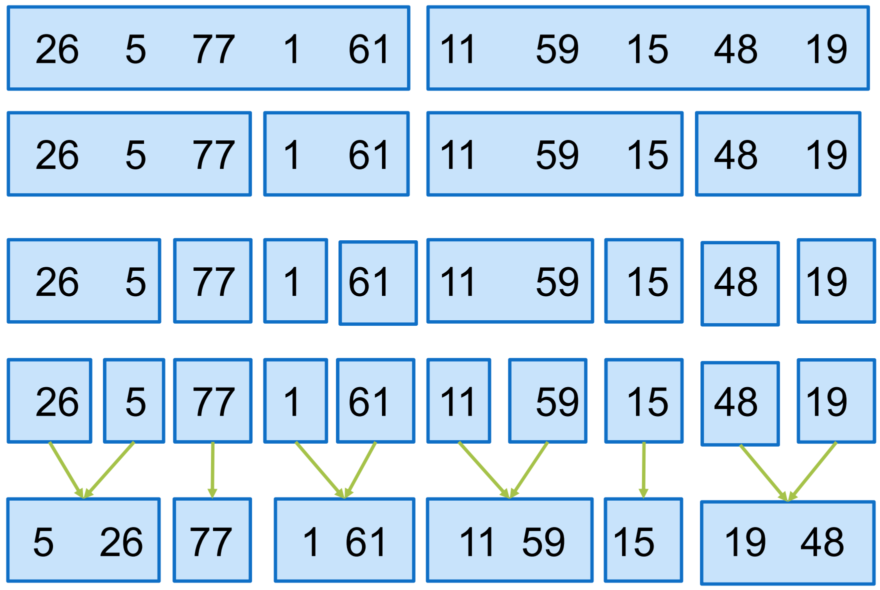
Properties

- Time complexity
  - Number of merge pass: O(logn)
  - Time complexity of merge pass: O(n)
  - Time complexity = O(nlogn)
- Require additional storage to store merged results
- Stable sort

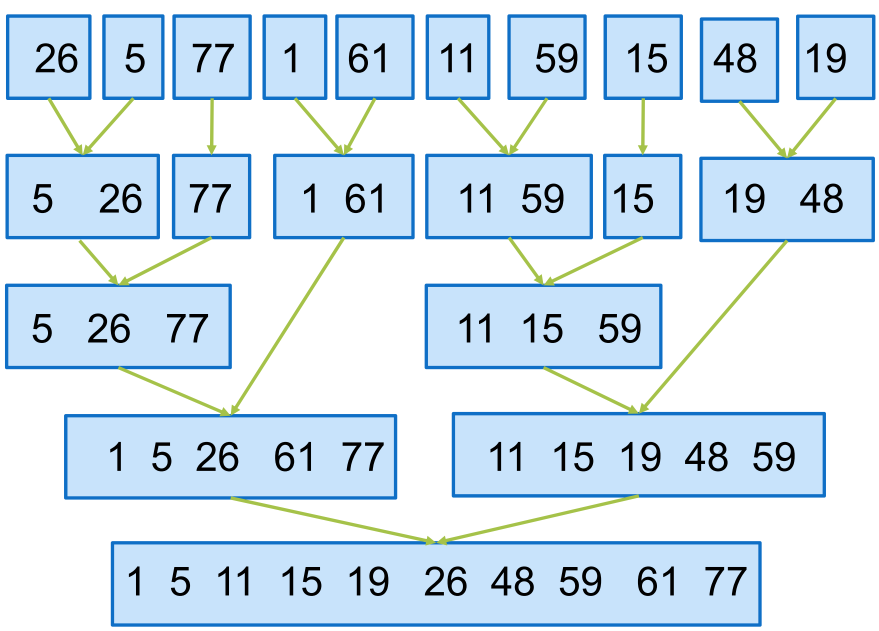
Recursive Merge Sort

- 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

Example of Recursive Merge Sort



Example of Recursive Merge Sort (Contd.)



Recursive Merge Sort (codes)

```
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.
}
```

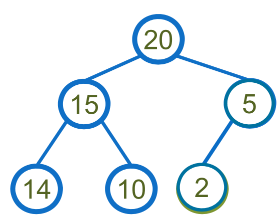


```
tamplate <class T>
int ListMerge(T* a, int* link, const int start1, const int
start2)
{// merge two sorted lists, starting from start1 and start2.
 // link[0] is a temporary head, stores the head of merged list.
 // iRsults records the last element of currently merged list.
 int iResult = 0;
 for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

Heap Sort

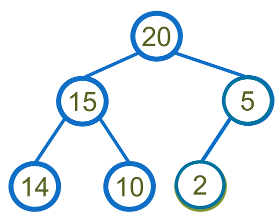
Recap

- Heap: Ordered binary tree
  - A complete binary tree
- Max heap: parent > child
  - Can adopt "**Array Representation**"
    - Since it is a complete binary tree
  - Let node i be in position i (array[0] is empty)
    - Parent(i) = i / 2 if i ≠ 1. If i=1, i is the root and has no parent
    - leftChild(i) = 2i if 2i ≤ n. If 2i > n, the i has no left child.
    - rightChild(i) = 2i+1 if 2i+1 ≤ n, if 2i+1 > n, the i has no right child



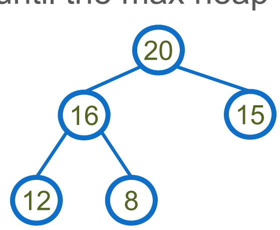
Recap: Insert in Max Heap

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



Recap: Delete in Max Heap

- Priority Queues
  - The element to be deleted is the one with highest priority
- In priority queues
  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)



Heap Sort

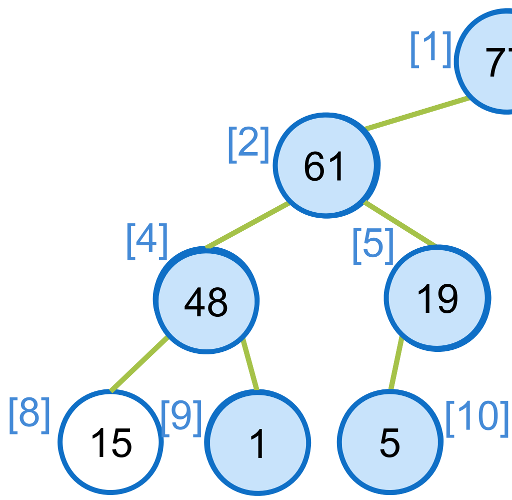
- Utilize the max-heap structure
  - The insertion and deletion could be done in O(logn)
- Build a max-heap using *n* records, insert each record one by one (O(nlogn))
- Iteratively remove the largest record (the root) from the max-heap (O(nlogn))
- Not a stable sort

Heap Sort (codes)

```
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])
  }
}
```

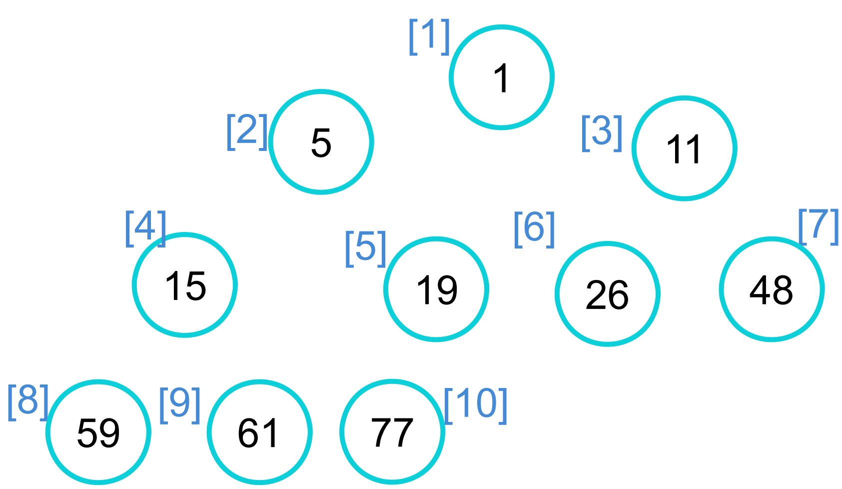
Running Example for Heap Sort

26 5 77 1 61 11 59 15 48 19



Running Example for Heap Sort

1 5 11 15 19 26 48 59 61 77



Sorting with Several Keys

Sorting a Deck of Cards

- A list of records with respect to the keys $K^1, K^2, \ldots, K^r$
  - iff for every pair of records i and j, i < j and $(K_i^1, K_i^2, \ldots, K_i^r) \le (K_j^1, K_j^2, \ldots, K_j^r)$
- 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 ♠

Sorting Approaches

- 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
- 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
  - Using a stable sort with respect to K^1 and obtain 4 "piles"
  - Merge piles by placing the piles on top of each other

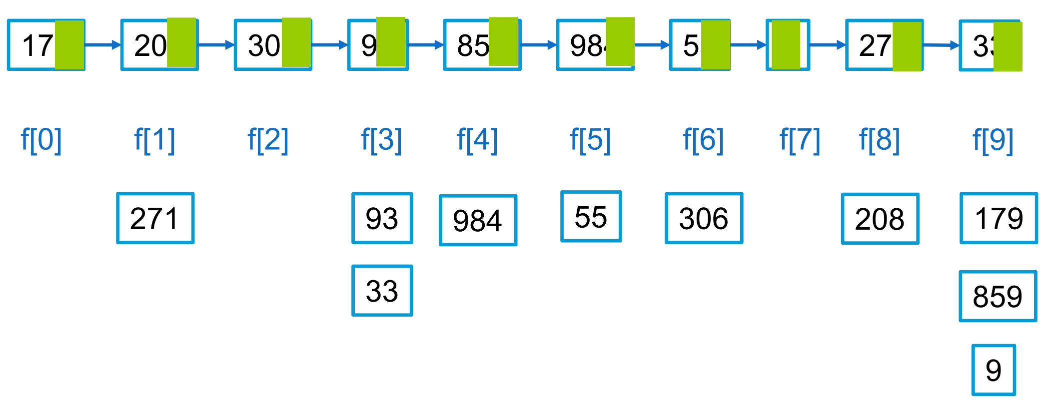
Bin Sort (Bucket Sort)

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

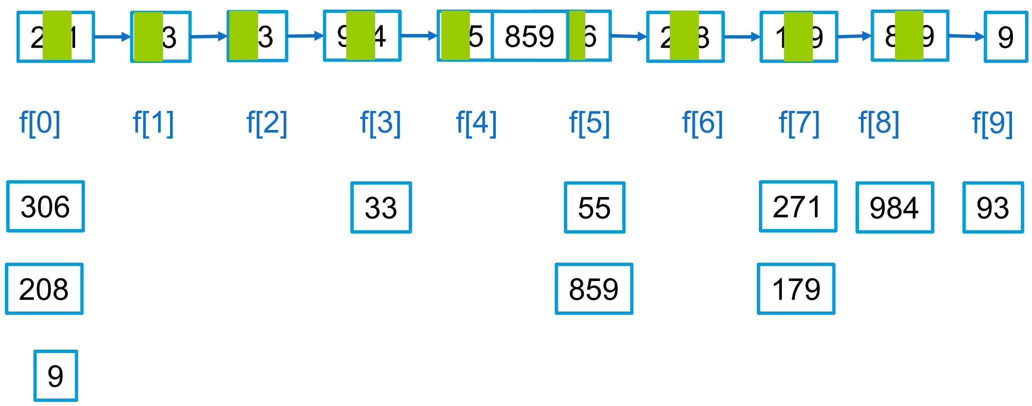
Radix Sort

- Decompose the key (number) into subkeys using some **radix** (base) **r**
- Create **r-1** buckets
- Apply bin sort with MSD or LSD order
- Suitable to sort numbers with large value range

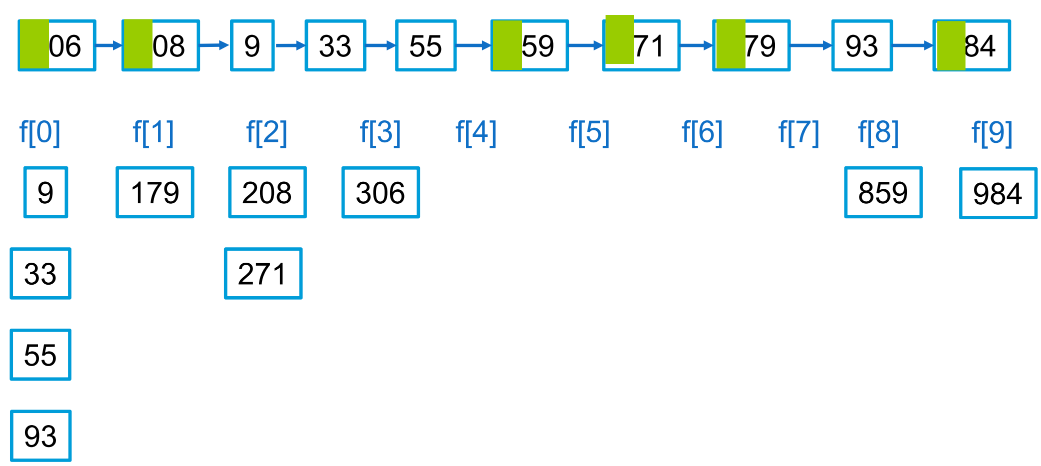
Radix Sort Example (Pass 1)



Radix Sort Example (Pass 2)



Radix Sort Example (Pass 3)



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

LSB Radix Sort (codes)

```
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 jth 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…
 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;
   }
```


LSB Radix Sort (codes)

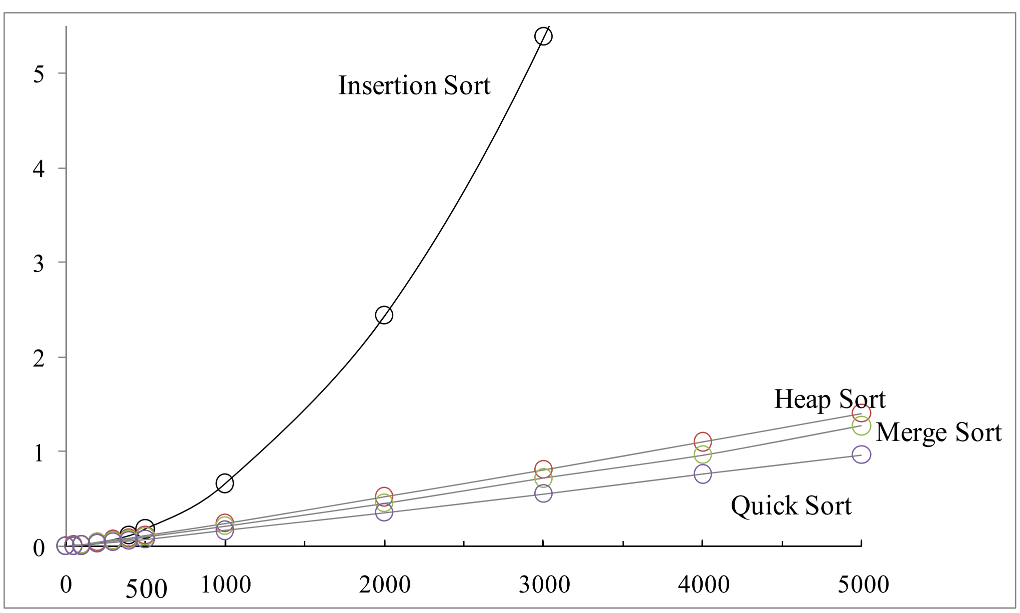
```
for (j = 0; !f[j]; j++); // find the 1st non-empty bin
   first = f[j];
   int last = e[j];
   for (int k = j + 1; k < r; k++){ // link the rest of bins
     if (f[k]) {
        link[last] = f[k];
        last = e[k];}
   }
   link[last] = 0;
 }
 return first;
}
```

Internal Sorting Summary

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

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

Internal Sorting Summary



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

External Sort

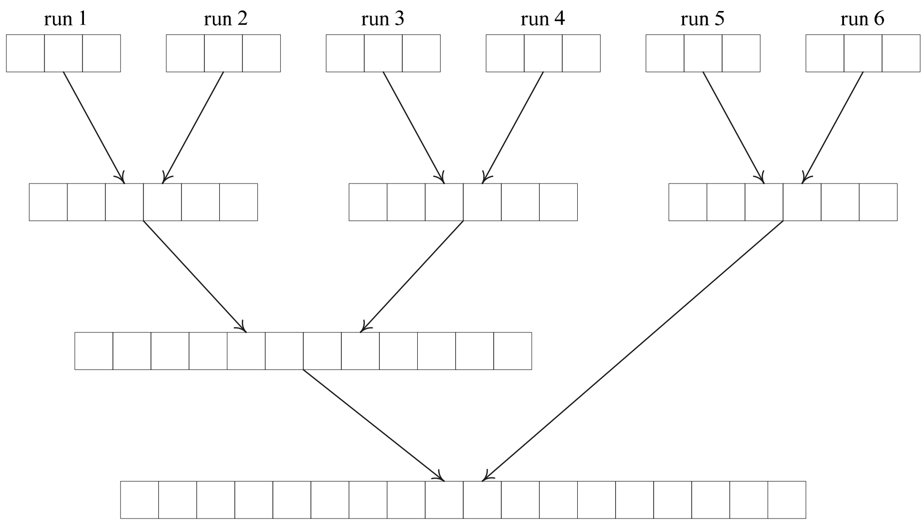
External Sort

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

External Sorting Algorithm

- Insertion sort, Quick sort, Heap sort
- Merge sort
  - Segments (runs) of input lists sorted using an internal sort
  - The runs generated in phase one are merged together following the **merge-tree** pattern
- Why merge sort?
  - Sublists could be sorted independently and merged later
  - During the merging, only the leading records of the two runs needed to be loaded in memory

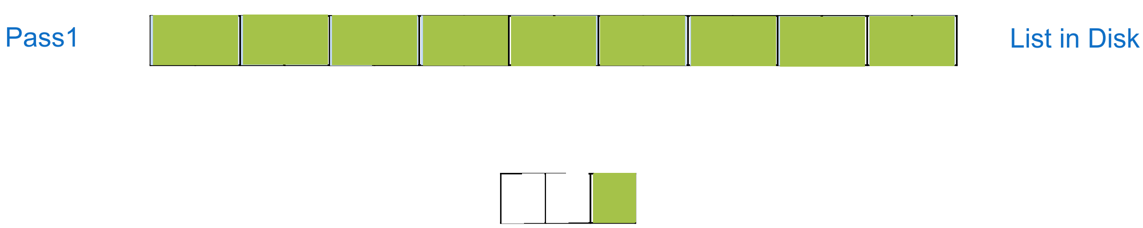
Runs & Merge Tree



Merge tree

Running Example for External Sorting

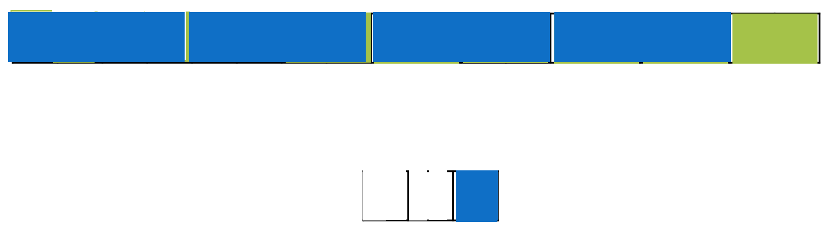
- Problem:
  - Internal memory: 750 records
  - Block size: 250 records
  - List to be sorted: 4500 (250*18) records
- To merge R1 and R2:
  - The first blocks of R1 and R2 are read into input buffers
  - The merged data is written to output buffer
  - Output buffer full → write onto disk
  - Input buffer empty → read from the new block



Running Example for External Sorting

- Problem:
  - Internal memory: 750 records
  - Block size: 250 records
  - List to be sorted: 4500 (250*18) records
- To merge R1 and R2:
  - The first blocks of R1 and R2 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

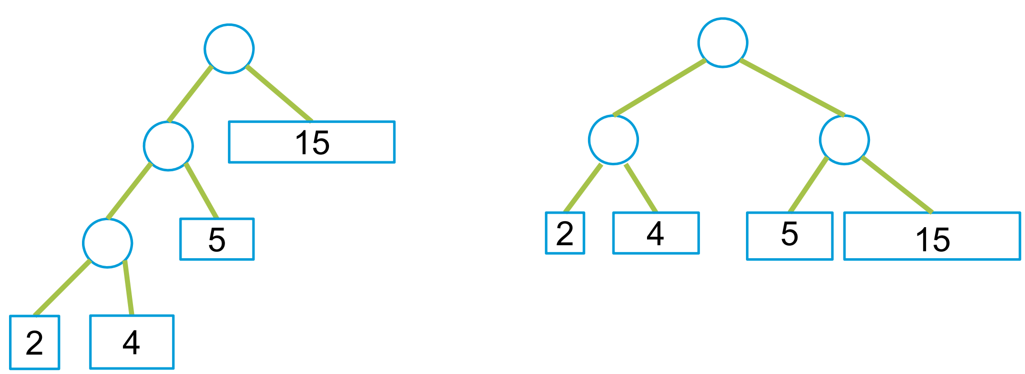
Pass2



List in Disk

Optimal Merging of Runs

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



Weighted External Path Length

- The total number of merge 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

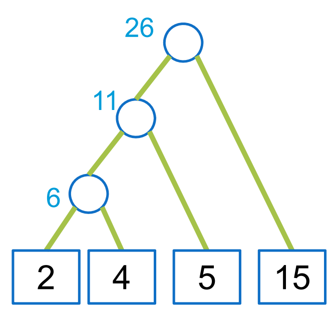
- How to build a merge tree such that the total cost is minimized?

Weighted External Path Length

- Sort runs using its size

2	4	5	15

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



Message Encoding

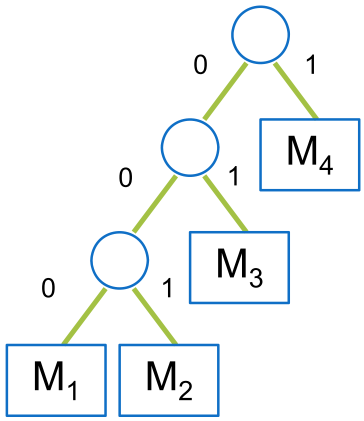
Message Encoding

- Given a set of messages $\{M_1, M_2, \ldots, M_i\}$
- How do we encode each M_i using a binary code such that each code is unique?

	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

Huffman Codes

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



Decode tree

	Huffman Codes
M_1	000
M_2	001
M_3	01
M_4	1

Huffman Codes

- Cost of decoding a code word is proportional to the number of bits in the code
- Assume the frequency of a message M_i been transmitted is q_i, the total cost is:

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

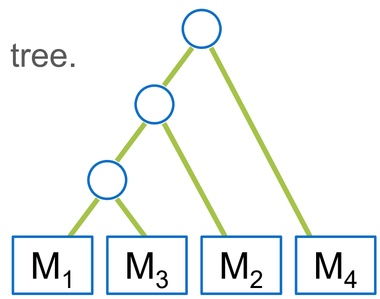
- How do we construct a decode tree such that the transmission cost is minimized?

Weighted External Path Length

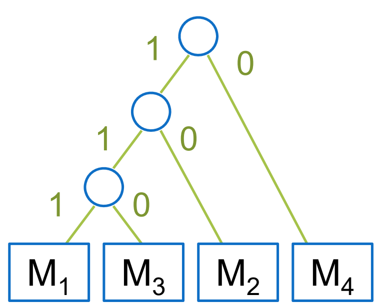
- Sort the message according to q_i

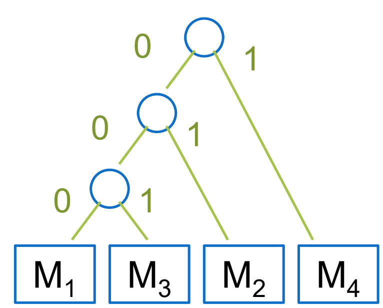
M_1	M_3	M_2	M_4
1/7	1/7	2/7	3/7

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



Message Encoding





	Huffman Codes
M_1	111
M_2	10
M_3	110
M_4	0

	Huffman Codes
M_1	000
M_2	01
M_3	001
M_4	1