1. Table of Contents
2. Introduction
3. Bubblesort
4. Bubblesort Complexity
5. Bubblesort Complexity (cont)
6. Selection Sort
7. Selection Sort Complexity
8. Duplex Selection Sort
9. Duplex Selection Sort (cont)
10. Comparison Order Analysis
11. Comparison Order Analysis (cont)
12. Time Analysis
13. Minimal Comparison
14. Comparison Tree Argument
15. Minimal Order : Comparison Sorting
16. Quicksort: partitioning
17. Quicksort: find pivot
18. Recursive Quicksort
19. Iterative Quicksort
20. Iterative Quicksort Code
21. Quicksort Efficiency
22. Sorting Considerations
23. Better than n log(n)?
24. BinSort
25. BinSort Implementation
Definitions and Terminology:
- Internal sorts hold all data in RAM
- External sorts use Files
- Ascending Order:
  † Low to High
- Descending Order:
  † High to Low
- Stable Sort:
  † Maintains the relative order of equal elements, in situ.
  † Desirable if list is almost sorted or if items with equal values are to also be ordered on a secondary field.

Program efficiency
- Overall program efficiency may depend entirely upon sorting algorithm => clarity must be sacrificed for speed.

Sorting Algorithm Analysis
- Performed upon the “overriding” operation in the algorithm:
  † Comparisons
  † Swaps

```c
void swap( int& x, int& y ) {
    int tmp = x;
    x = y;
    y = tmp;
}
```
Behavior

- *Bubble* elements down (up) to their location in the sorted order.

```c
for (i = 0; i < n-1; i++)
    for (j = n-1; j > i; j--)
        if (A[j] < A[j-1])
            swap(A[j], A[j-1]);
```

Graphical Trace

- **Starting**
- **Working**
- **Working**
- **Finished**
13. Sorting

```c
void BubbleSort(int A[], const int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = n-1; j > i; j--)
            if (A[j] < A[j-1])
                swap(A[j], A[j-1]);
}
```

Time Analysis

if-statement:
    time 2 for the condition (and subtraction) + time 4 for the swap (3 assigns + setup), so 5

inner for-loop:
    body executed for j-values from n-1 down to i+1, or n-i-1 times
    loop body is time 6 for if-stmt + time 2 for test and update

    \[
    \sum_{j=1}^{n-i-1} 8 = 8(n - i - 1)
    \]

outer for-loop:
    body executed for i-values from 0 up to n-2 (or 1 to n-1)
    loop body is time for inner for-loop + time 3 for loop test and update

So counting the initialization & pre-tests of the for-loop, the overall complexity is:

\[
3 + \sum_{i=1}^{n-1} \left( 3 + 2 + 8(n - i - 1) \right) = 3 + \sum_{i=1}^{n-1} \left( 8n - 8i - 3 \right)
\]

\[
= 3 + 8n(n-1) - \frac{8}{2} (n-1)(n-2) - 3(n-1)
\]

which is clearly O(n^2).
void BubbleSort(int A[], const int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = n-1; j > i; j--){
            if (A[j] < A[j-1])
                swap(A[j], A[j-1]);
        }
}

Swaps and Compares Analysis

if-statement:
    1 compare and in worst case, 1 swap

inner for-loop:
    body executed for j-values from n-1 down to i+1, or n-i-1 times
    each execution of body involves 1 compare and up to 1 swap

outer for-loop:
    body executed for i-values from 0 up to n-2 (or 1 to n-1)
    each execution of body involves n-i-1 compares and up to n-i-1 swaps

So in the worst case, the number of swaps equals the number of compares, and is:

\[ \sum_{i=1}^{n-1} (n-i-1) = n(n-1) - \frac{1}{2} (n-1)(n-2) - (n-1) \]

which is clearly \( O(n^2) \).
Selection Sort

Behavior

- In the $i^{th}$ pass, select the element with the lowest value among $A[i]$, ..., $A[n-1]$, & swap it with $A[i]$.
- Results after $i$ passes: the $i$ lowest elements will occupy $A[0]$, ..., $A[i]$ in sorted order.

```c
for (Begin = 0; Begin < Size - 1; Begin++) {
    SmallSoFar = Begin;
    for (Check = Begin + 1; Check < Size; Check++) {
        if (aList[Check] < aList[SmallSoFar])
            SmallSoFar = Check;
    }
    swap(aList[Begin], aList[SmallSoFar]);
}
```

Graphical Trace

Working

Working

Finished
void SelectionSort(int aList[], const int Size) {
    int Begin, SmallSoFar, Check;

    for (Begin = 0; Begin < Size - 1; Begin++) {
        SmallSoFar = Begin;
        for (Check = Begin + 1; Check < Size; Check++) {
            if (aList[Check] < aList[SmallSoFar])
                SmallSoFar = Check;
        }
        swap(aList[Begin], aList[SmallSoFar]);
    }
}

Swaps and Compares Analysis

if-statement: 1 compare

inner for-loop:
    body executed n-i-1 times (i is Begin and n is Size)
    each execution of body involves 1 compare and no swaps

outer for-loop:
    body executed n-1 times
    each execution of body involves n-i-1 compares and 1 swap

So in the worst case, the number of swaps is n – 1, and the number of compares is:

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

which is clearly O(n²) and the same as for BubbleSort.
Duplex Selection Sort

13. Sorting 8

Min / Max Sorting
- algorithm passes thru the array locating the min and max elements in the array \( A[i], \ldots, A[n-i+1] \). Swapping the min with \( A[i] \) and the max with \( A[n-i+1] \).
- Results after the \( i \)th pass: the elements \( A[1], \ldots, A[i] \) and \( A[n-i+1], \ldots, A[n] \) are in sorted order.

**Unsorted Array**

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>77</td>
<td>63</td>
<td>58</td>
<td>42</td>
<td>37</td>
<td>31</td>
<td>26</td>
<td>19</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

**After 1st Pass**

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>77</td>
<td>63</td>
<td>58</td>
<td>42</td>
<td>37</td>
<td>31</td>
<td>26</td>
<td>19</td>
<td>95</td>
<td></td>
</tr>
</tbody>
</table>

**After 3rd Pass**

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>19</td>
<td>26</td>
<td>58</td>
<td>42</td>
<td>37</td>
<td>31</td>
<td>63</td>
<td>77</td>
<td>95</td>
<td></td>
</tr>
</tbody>
</table>

5 passes to required sort the above array = \( N / 2 \).
**Duplex Selection Sort (cont)**

**Code**

```c
void DuplexSelectSort(raytype ray, int start, int finish) {
    int low, high, min, max, small, large, minmax;
    Item tmp;

    low = start;  high = finish;

    while (low < high) {
        small = ray[low];  min = low;
        large = ray[high];  max = high;

        //search for smallest & largest
        for (minmax = low; minmax <= high; minmax++) {
            if (ray[minmax] < small) {
                min = minmax;
                small = ray[minmax];
            } else if (ray[minmax] > large) {
                max = minmax;
                large = ray[minmax];
            }
        }  // for minmax

        //check for swap interference
        if ((max == low) && (min == high)) {
            swap(ray[low], ray[high]);
        } else if (max == low) {
            swap(ray[max], ray[high]);
            swap(ray[low], ray[min]);
        } else {
            swap(ray[min], ray[low]);
            swap(ray[max], ray[high]);
        }
        low++;
        high--;
    }  // while
}
```

*Recursive implementation: slide 9.14*
Comparison $O$(DuplexSelectSort)

- Outer Loop: WHILE $i$ loop
  
  loop limits               shifted limits
  $= 0 \ldots \frac{N}{2} - 1$               $= 1 \ldots \frac{N}{2}$

- Assume subset of array to sort is from $1 \ldots N$
  
  $= 1 \ldots \frac{N}{2}$ (i.e. start ... finish)

- Inner Loop: FOR $j$ loop

<table>
<thead>
<tr>
<th>Pass (i)</th>
<th>loop limits</th>
<th>shifted limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>$1 \ldots N$</td>
<td>$= 1 \ldots N$</td>
</tr>
<tr>
<td>2nd</td>
<td>$2 \ldots N-1$</td>
<td>$= 1 \ldots N-2$</td>
</tr>
<tr>
<td>3rd</td>
<td>$3 \ldots N-2$</td>
<td>$= 1 \ldots N-4$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ith iteration</td>
<td>$i \ldots N-(i-1)$</td>
<td>$= 1 \ldots N-(i-1)*2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$\frac{N}{2}-1$</td>
<td>$= \frac{N}{2}-1 \ldots N - (\frac{N}{2}-1)$</td>
<td>$= 1 \ldots 4$</td>
</tr>
<tr>
<td>$\frac{N}{2}$</td>
<td>$= \frac{N}{2} \ldots N - (\frac{N}{2}-1)$</td>
<td>$= 1 \ldots 2$</td>
</tr>
</tbody>
</table>
Comparison Order Analysis (cont) 13. Sorting 11

Worst Case:

2 Comparisons on each element

\[
\begin{align*}
\sum_{i=1}^{N/2} \left[ \sum_{j=i}^{N-(i-1)} 2 \right] &= \sum_{i=1}^{N/2} \left[ \sum_{j=1}^{N-2(i-1)} 2 \right] \\
&= \frac{N}{2} \left[ \frac{N}{2} + \frac{N}{2} \right] \\
&= \frac{N^2}{2} + N 
\end{align*}
\]

expanding yields:

\[
= 2 \left[ N+N-2+N-4+ \ldots +6+4+2 \right] 
\]

summing the arithmetic sequence yields:

\[
= 2 \left[ \frac{1}{2} \left( \frac{N}{2} \left[ N + 2 \right] \right) \right] 
\]

\[
= \frac{N^2}{2} + N \in O(N^2) 
\]

expanding yields:

\[
= \frac{N}{2} \left( \frac{N}{2} + 1 \right) - \frac{N}{2} 
\]

distributing yields:

\[
= 2 \left( \frac{N}{2} \right) - 4 \left( \frac{N^2}{4} + \frac{N}{2} - \frac{N}{2} \right) 
\]

\[
= N^2 - 4 \left( \frac{N^2}{8} + \frac{N}{4} - \frac{N}{2} \right) 
\]

\[
= N^2 - \frac{N^2}{2} - N + 2N 
\]

\[
= \frac{N^2}{2} + N \in O(N^2) 
\]
Assume: \( \text{Start} = 1 \) \( \text{Finish} = N \)

Analysis: (worst case)
- Initialization of High and Low = 2
- While Loop
  - pretest condition = 1
  - loop body executes \( \frac{N}{2} \) times (i from 1 to \( \frac{N}{2} \))
  - time cost is 7 (for while condition + assignments + increments) +
  - time for for-loop + time for swap code
- For-loop
  - initialization & pretest condition = 2
  - loop body executes \( N-2 \) \( (i-1) \) times
  - time cost is 2 (for test and increment) + time for if
  - time cost of if is 4 (two if-conditions + 2 assignments)
- Swap Code
  - time cost is 3 for first if-condition + 1 for second if-condition + cost of two swaps
  - 1 for function setup + time cost of a swap is 3 for assignments

\[
T(N) = 3 + \sum_{i=1}^{N/2} \left( (7 + 2) + \sum_{j=1}^{N-2(i-1)} 6 + (3 + 1 + 4 + 4) \right)
\]

\[
= 3 + \sum_{i=1}^{N/2} (21 + 6(N - 2(i - 1)))
\]

\[
= 3 + \sum_{i=1}^{N/2} (6N + 33 - 12i)
\]

\[
= \frac{3}{2} N^2 + \frac{27}{2} N + 3
\]

So Duplex Selection Sort is another \( O(N^2) \) sort algorithm.

But, the coefficient IS better than for BubbleSort.
13. Sorting

Comparison-Based Sorting
- Algorithms which compare element values to each other
- What is the minimum number of comparisons, required to sort N elements using a comparison-based sort?

Comparison Tree
- Binary tree (hierarchical graph ≤ 2 branches per node) which contains comparisons between 2 elements at each non-leaf node & containing element orderings at its leaf (terminal) nodes.
- Comparison Tree for 3 Elements

Any of the 3 elements (a, b, c) could be first in the final order. Thus there are 3 distinct ways the final sorted order could start.
- After choosing the first element, there are two possible selections for the next sorted element.
- After choosing the first two elements there is only 1 remaining selection for the last element.
- Therefore selecting the first element one of 3 ways, the second element one of 2 ways and the last element 1 way, there are 6 possible final sorted orderings = \(3 \times 2 \times 1 = 3!\)
Order Tree for sorting N Elements
- Any of the N elements (1 ... N) could be first in the final order. Thus there are N distinct ways the final sorted order could start.
- After choosing the first element, there are N-1 possible selections for the next sorted element.
- After choosing the first two elements there N-2 possible selections for the next sorted element, etc.
- Therefore selecting the first element one of N ways, the second element one of N-1 ways, etc., there are N * (N-1) * (N-2) * ... * 2 * 1 possible final sorted orderings which = N!

General Comparison Tree Sorting
- The comparison tree for N elements must have N! leaf nodes. Each leaf node contains one of the possible orderings of all of the N elements.
- Consider the previous comparison tree for 3 elements, all of the leaf nodes are at a depth of either 2 or 3 = \lfloor \log_2 3! \rfloor
- The comparison tree for 4 elements must contain 4! = 24 leaf nodes, all of which would be at a depth of either 4 or 5 = \lfloor \log_2 4! \rfloor

\[ \text{Depth:} \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \]

floor \( y = \lfloor x \rfloor \), \( y \) is the largest integer such that \( y \leq x \).
Minimal Order: Comparison Sorting

General Comparison Trees

- The comparison tree for N elements must contain N! leaf nodes, all of which would be at a depth > ⌊log₂N⌋
- The minimal number of comparisons required to sort a specific (unsorted) ordering is equal to the depth from the root to a leaf.

Since the depth of all leaf nodes is > ⌊log₂N⌋ in a comparison tree, the minimal number of comparisons to sort a specific initial ordering of N elements is > ⌊log₂N⌋
- Stirling’s Approximation for log₂(N!) can be used to determine a lower bound for log₂(N!) which is O(NlogN)
- No comparison based sorting algorithm can sort faster than

\[ O(N \log N) \]
QuickSort: partitioning

Algorithm
- Select an item in the array as the pivot key.
- Divide the array into two partitions: a left partition containing elements < the pivot key and a right partition containing elements > the pivot key.

Trace
Start with i and j pointing to the first & last elements, respectively.
Select the pivot (3): [3 1 4 1 5 9 2 6 5 8]
   R   L
Swap the end elements, then move l, r inwards.
   [8 1 4 1 5 9 2 6 5 3]
   L   R
Swap, and repeat: [2 1 4 1 5 9 8 6 5 3]
   L   R
Swap, and repeat: [2 1 1 4 5 9 3 6 5 3]
   R   L

Partition Function:

```c
int Partition(Item A[], int start, int end, Item pivot ){
    int L = start, R = end;
    do {
        swap( A[L] , A[R] );
        while (A[L] < pivot )  L++;
        while (!(A[R] < pivot))  R--;
    } while (R > L);
    return (L);
}
```
## 13. Sorting

### Pivoting

- Partitioning test requires at least 1 key with a value < that of the pivot, and 1 value > to that of the pivot.
- Therefore, pick the greater of the first two distinct values (if any).

```c
const int MISSING = -1;

int FindPivot(const Item A[], int start, int end) {
    Item firstkey; // value of first key found
    int pivot; // pivot index
    int k; // run right looking for other key

    firstkey = A[start];
    // return -1 if different keys are not found
    pivot = MISSING;
    k = start + 1;
    // scan for missing key
    while ((k <= end) && (pivot == MISSING))
    {
        if (firstkey < A[k]) // select key
            pivot = k;
        else if (A[k] < firstkey)
            pivot = start;
        else
            k++;
    }
    return pivot;
}
```

### Improving FindPivot

- Try and pick a pivot such that the list is split into equal size sublists, (a speedup that should cut the number of partition steps to about 2/3 that of picking the first element for the pivot).
  - Choose the middle (median) of the first 3 elements.
  - Pick k elements at random from the list, sort them & use the median.
- There is a trade-off between reduced number of partitions & time to pick the pivot as k grows.
13. Sorting

Quicksort Function (recursive)

```c
const int MISSING = -1;
void QuickSort( Item A[], int start, int end ) {
    // sort the array from start ... end
    Item pivotKey;
    int pivotIndex;
    int k; //index of partition >= pivot

    pivotIndex = FindPivot( A, start, end );
    if (pivotIndex != MISSING) {
        pivotKey = A[pivotIndex];
        k = Partition( A, start, end, pivotKey );
        QuickSort( A, start, k-1 );
        QuickSort( A, k, end );
    }
}
```

Average Case = \( O(N \log N) \)
- quicksort is based upon the intuition that swaps, (moves), should be performed over large distances to be most effective
- quicksort's average running time is faster than any currently known \( O(n \log_2 n) \) internal sorting algorithms (by a constant factor).
- For very small \( n \) (e.g., \( n \leq 16 \)) a simple \( O(n^2) \) algorithm is actually faster than Quicksort.
- When the sublist is small, use another sorting algorithm.

Worst Case = \( O(N^2) \)
- In the worst case, every partition might split the list of size \( j - i + 1 \) into a list with 1 element, and a list with \( j - i \) elements.
- A partition is split into sublists of size 1 & \( j-i \) when one of the first two items in the sublist is the largest item in the sublist which is chosen by findpivot.
- When will this worst case partitioning always occur?
Iterative Quicksort

Iterative Conversion
- Iterative implementation requires using a stack to store the partition bounds remaining to be sorted.
- Assume a stack implementation of elements consisting of two integers:

```
struct StackItem {
    int low, hi;
};
```

Partitioning
- At the end of any given partition, only one subpartition need be stacked.
- The second subpartition (equated to the second recursive call), need not be stacked since it is immediately used for the next subpartitioning.

Stacking
- The order of the recursive calls, (i.e., the sorting of the subpartitions) may be made in any order.
- Stacking the larger subpartition assures that the size of the stack is minimized, since the smaller subpartition will be further divided less times than the larger subpartition.
Quicksort Function (iterative)

```c
void QuickSort( Item A[], int start, int end ) {
    // sort the array from start ... end
    Item pivotKey;
    int pivotIndex, tmpBnd;
    int k; // index of partition >= pivot
    StackItem parts;
    Stack subParts;

    parts.low = start; parts.hi = end;
    subParts.Push( parts );

    while ( ! subParts.Empty() ) {
        parts = subParts.Pop();

        while ( parts.hi > parts.low ) {
            pivotIndex = FindPivot( A, parts.low, parts.hi );
            if (pivotIndex != MISSING) {
                pivotKey = A[pivotIndex];
                k = Partition( A, parts.low, parts.hi , pivotKey );
                // push the larger subpartition
                if ( (k-parts.low) > (parts.hi-k) ) { // stk low part
                    tmpBnd = parts.hi;
                    parts.hi = k-1;
                    subParts.Push( parts );
                    parts.low = k; // set current part to upper part
                    parts.hi = tmpBnd;
                } // end if
                else { // stack upper (larger) part
                    tmpBnd = parts.low;
                    parts.low = k;
                    subParts.Push( parts );
                    parts.low = tmpBnd; // set current part to low part
                    parts.hi = k-1;
                } // end else
            } // end if
        } // end while
    } // end while
} // end QuickSort
```
Minor Improvements

- All function calls should be replaced by inline code to avoid function overhead.
- Current partition bounds should be held in register variables.
Sorting Considerations

General

- With large data records, swap pointers instead of copying records:

There’s a tradeoff here between storage cost and time. There’s also a time versus time tradeoff. We’re accepting the cost of additional pointer dereferences to avoid the cost of some data copying.

- Carefully investigate the average data arrangement in order to select the optimal sorting algorithm.
- No one algorithm works the best in all cases.
Special Techniques for Special Cases

- If sort key (member) consists of consecutive (unique) N integers they can be easily mapped onto the range 0 .. N-1 & sorted.
- If the N elements are initially in array A, then:

```c
Item A[], B[];
for (int i = 0; i < N; i++ )
   B[A[i].GetKey() % N] = A[i];
```

takes O(n) time.
- Requires exactly 1 record with each key value!
- Of course, this is a very special circumstance…
- Special case of Bin Sorting. (If integers are not consecutive, but within a reasonable range, used bit flags can be used to denote empty array slots.)
BinSort is a somewhat more general O(N) sort technique.

Assume we need to sort an array of integers in the range 0-99:

\[
\begin{array}{cccccccccc}
34 & 16 & 83 & 76 & 40 & 72 & 38 & 80 & 89 & 87 \\
\end{array}
\]

Assume we have an array of 10 linked lists (bins) for storage.

First make a pass through the list of integers, and place each into the bin that matches its 1’s digit.

Then, make a second pass, taking each bin in order, and place each integer into the bin that matches its 2’s digit.

\[
\begin{array}{|c|c|c|}
\hline
\text{Bin} & \begin{array}{ll}
0: & 40 \ 80 \\
1: & \\
2: & 72 \\
3: & 83 \\
4: & 34 \\
5: & \\
6: & 16 \ 76 \\
7: & 87 \\
8: & 38 \\
9: & 89 \\
\end{array} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Bin} & \begin{array}{ll}
0: & \\
1: & 16 \\
2: & \\
3: & 34 \ 38 \\
4: & 40 \\
5: & \\
6: & \\
7: & 72 \ 76 \\
8: & 80 \ 83 \ 87 \ 89 \\
9: & \\
\end{array} \\
\hline
\end{array}
\]

Now if you just read the bins, in order, the elements will appear in ascending order. Each pass takes O(N) work, and the number of passes is just the number of digits in the largest integer in the original list.

That beats QuickSort, but only in a somewhat special case.
void BinSort(int Data[], int numData, LinkList Bin[]) {

    // First pass:
    for (int Idx = 0; Idx < numData; Idx++) {
        int Digit1 = Data[Idx] % 10;
        Bin[Digit1].gotoTail();
        Bin[Digit1].Insert(Item(Data[Idx])); //append to end
    }

    // Second pass:
    LinkList Bin2[NumBins];
    for (Idx = 0; Idx < 10; Idx++) {
        Bin[Idx].gotoHead();
        while (Bin[Idx].inList()) {
            int currValue =
                Bin[Idx].getCurrentData().getValue();
            int Digit2 = (currValue / 10) % 10;
            Bin2[Digit2].gotoTail(); //append to end
            Bin2[Digit2].Insert(Item(currValue));
            Bin[Idx].Advance();
        }
    }
    LinearizeBins(Bin2, Data);
}

void LinearizeBins(LinkList Bin[], int Target[]) {

    int Tidx = 0;

    for (int Idx = 0; Idx < 10; Idx++) {
        Bin[Idx].gotoHead();
        while (Bin[Idx].inList()) {
            Target[Tidx] =
                Bin[Idx].getCurrentData().getValue();
            Tidx++;
            Bin[Idx].Advance();
        }
    }
}