Sorting - 3

CS 202 – Fundamental Structures of Computer Science II
Bilkent University
Computer Engineering Department

MergeSort - Continued

```
template <class Comparable>
void mergeSort( vector<Comparable> & a )
{
    vector<Comparable> tmpArray( a.size( ) );

    mergeSort( a, tmpArray, 0, a.size( ) - 1 );
}

template <class Comparable>
void mergeSort( vector<Comparable> & a,
                vector<Comparable> &  tmpArray, int left, int right )
{
    if( left < right )
    {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}
```

```
template <class Comparable>
void merge( vector<Comparable> & a, vector<Comparable> & tmpArray,
      int leftPos, int rightPos, int rightEnd )
{
    int leftEnd       = rightPos - 1;
    int tmpPos        = leftPos;
    int numElements = rightEnd - leftPos + 1;

    // Main loop
    while( leftPos <= leftEnd && rightPos <= rightEnd )
       if( a[ leftPos ] <= a[ rightPos ] )
          tmpArray[ tmpPos++ ] = a[ leftPos++ ];
       else
          tmpArray[ tmpPos++ ] = a[ rightPos++ ];

    while( leftPos <= leftEnd )    // Copy rest of first half
       tmpArray[ tmpPos++ ] = a[ leftPos++ ];

    while( rightPos <= rightEnd )  // Copy rest of right half
       tmpArray[ tmpPos++ ] = a[ rightPos++ ];

    // Copy tmpArray back
    for( int i = 0; i < numElements; i++, rightEnd-- )
       a[ rightEnd ] = tmpArray[ rightEnd ];
}
```


Analysis of MergeSort

- Mergesor() is a recursive routine
- There ia general technique to analyze recursive routines
- First we need to write down a *recurrence relation* that expresses the cost of procedure.
  - T(N) = ….
- Assume *the input size to the MergeSort,* N, is a power of 2.

Analysis of MergeSort

- Lets compute the running time
- If N = 1
  - The cost of mergesort is O(1). We will denote this as 1 in T(N) formula.
- If (N > 1)
  - The mergesort algorithm cosists of:
    - Two mergesorts on input of N/2. Running time = T(N/2)
    - A merge routing that is linear with respect to input size. O(N).
- Then: $T(N) = 2T(N/2) + 1$

Analysis of MergeSort

- We need to solve this recurrence relation!
- One way is like the following:
  - The idea is to expand each recursive part by substitution.
- $T(N) = 2\,T(N/2) + 1$ (1)
- $T(N/2) = 2T(N/4) + 1$ (2)
- Substitude T(N/2) in formula (1)
  - $T(N) = 2\,(2\,T(N/4) + 1) + N$

Analysis of MergeSort

- Continue doing this
  - T (N) = 2 (2 (2 T(N/8) + N) + N) + N
    = 2^3T(N/2^3) + 3
  - In termination case we have T(1) = 1
  - For having T(1) = T(N/2^k), we should have k = logN
- k = logN
  - T(N) = 2^kT(N/2^k) + kN
  - T(N) = 2^{logN}T(N/2^{logN}) + NlogN
  - T(N) = NT(1) + NlogN
  - *T(N) = N +N logN*

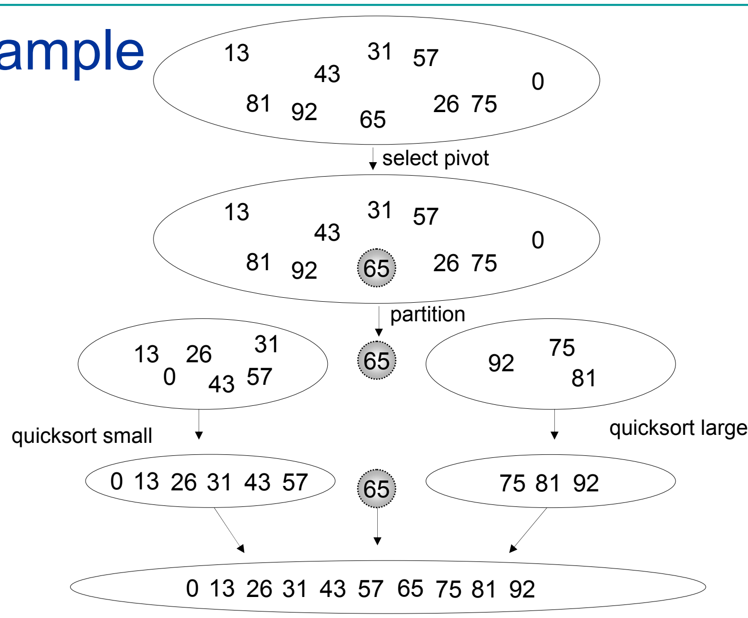
QuickSort

- Fastest known sorting algorithm in practice.
  - For in-memory sorting.
- O(NlogN) average running time
- O(N^2) worst-case performance, which can be very rare.
- The inner loop in algorithms is very optimized.

QuickSort - Algorithm

- Input: *S* – an array of elements of size *N*.
- Output: *S* – in sorted order.
  1. If the number of elements in *S* is 0 or 1, then return.
  2. Pick any element *v* in *S*. This is called the ***pivot***
  3. Partition *S-{v}* into two disjoint groups:
     S_1 = {*x* in *S-{v}* | *x* <= *v*} and
     S_2 = {*x in S-{v}* | *x* >= *v*}
  4. Return {quicksort(S_1)} followed by *v* followed by {*quicksort*(S_2)}

Example



- Partitioning can be performed over the *same* array.
- After partition, the two parts may be equal sized.
- Choosing the pivot value is important to have
  - Both parts S_1 and S_2 to have close to equal sizes.

Picking the Pivot

- Wrong way:
  - Choose the first element of array
    - What is the array was sorted!
- A safe method
  - Pick it up randomly among the elements of array
  - Depends on the quality of random number generator
- A good method:
  - Pick the *median* of three elements:
    - First elements
    - Last element
    - Middle element (lowerbound((first+last)/2)
  - Definition: M*edian* of N elements is the *lowerbound(N/2)th* largest element.
  - Example: Median of {7, 3, 4} is 4.

Partitioning Strategy

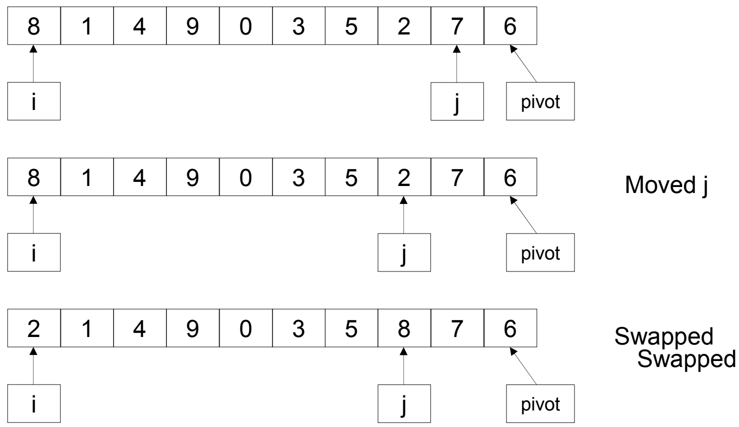
- Requires O(N) running time.

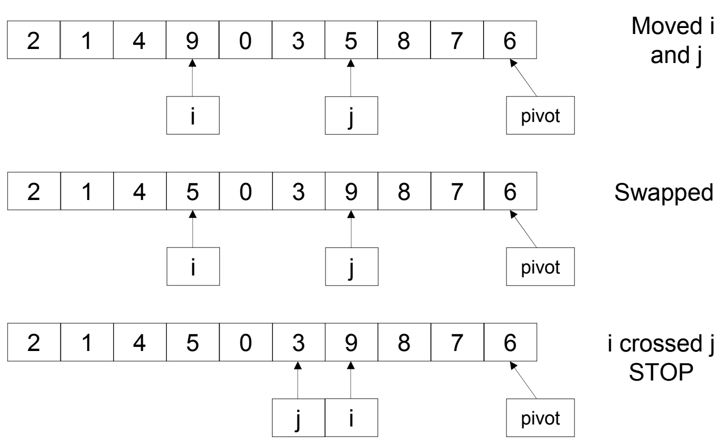
1. First find the pivot.
2. Then swap the pivot with the last element
3. Then do the following operations on elemente from *first* to *last-1* (last contains the pivot)
   - *Move all element smaller than pivot to the left of array*
   - *Move all element greater than pivot to the right of array*

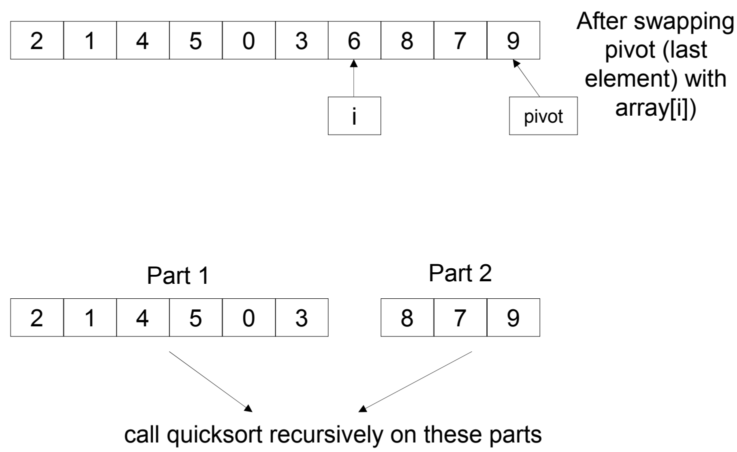
Partitioning Strategy

- For Step 3:
  - Keep two index counters: i and j.
  1. Initialize i to *first* and j to *last-1*.
  2. While i is smaller or equal to j do
     1. Move i towards right until array[i] > pivot
     2. Move j towards left until array[i] < pivot.
     3. Swap array[i] and array[j]

Example


8 1 4 9 0 3 5 2 7 6
i
j
pivot
8 1 4 9 0 3 5 2 7 6
Moved j
i
j
pivot
2 1 4 9 0 3 5 8 7 6
Swapped
Swapped
i
j
pivot


2 1 4 9 0 3 5 8 7 6
Moved il
and j
i
j
pivot
2 1 4 5 0 3 9 8 7 6
Swapped
i
j
pivot
2 1 4 5 0 3 9 8 7 6
i crossed j
STOP
j
i
pivot



QuickSort Code

```
template <class Comparable>
      void quicksort( vector<Comparable> & a )
      {
          quicksort( a, 0, a.size( ) - 1 );
      }

template <class Comparable>
const Comparable &median3( vector<Comparable> & a, int left, int right )
{
         int center = ( left + right ) / 2;
         if( a[ center ] < a[ left ] )
            swap( a[ left ], a[ center ] );
         if( a[ right ] < a[ left ] )
            swap( a[ left ], a[ right ] );
         if( a[ right ] < a[ center ] )
            swap( a[ center ], a[ right ] );

         swap( a[ center ], a[ right - 1 ] ); // Place pivot at position right - 1
         return a[ right - 1 ];
      }
```

```
template <class Comparable>
void quicksort( vector<Comparable> & a, int left, int right )
{
   if( left + 10 <= right )
   {
      Comparable pivot = median3( a, left, right );
       // Begin partitioning
      int i = left, j = right - 1;
      for( ; ; )
      {
         while( a[ ++i ] < pivot ) { }; // move i to right
         while( pivot < a[ --j ] ) { };  // move j to left
         if( i < j )
            swap( a[ i ], a[ j ] );   // swap array[i] with array[j]
         else
            break;
       }

       swap( a[ i ], a[ right - 1 ] );  // Restore pivot – put pivot at ith position

       quicksort( a, left, i - 1 );      // Sort small elements – recursive call
       quicksort( a, i + 1, right );    // Sort large elements
   }
   else  // Do an insertion sort on the subarray if array size  is smaller than 10
       insertionSort( a, left, right );
}
```


Analysis of Quicksort

- It is a recursive algorithm like mergesort.
- We will again use recurrence relations
- We will analyze of 3 cases
  - Worst case
  - Best case
  - Average case

Analysis of Quicksort

- For N=1 or N=0
  - T(N) = 1
- For (N>1)
  - Running time T(N) is equal to the running time of the two recursive calls plus the linear time spent in partitioning
  - $T(N) = T(i) + T(N-i-1)+cN$,
    where i is the number of elements in the first part S_1

Worst Case (i=0) Analysis

The pivot is the smallest element all the time. $i = 0$

$$\begin{aligned} T(N) &= T(N-1) + cN, \quad N > 1 \\ T(N-1) &= T(N-2) + c(N-1) \\ T(N-2) &= T(N-3) + c(N-2) \\ &\vdots \\ T(2) &= T(1) + c(2) \end{aligned}$$

Adding them all yields:

$$T(N) = T(1) + c\sum_{i=2}^{N} i = O(N^2)$$

Best Case (i ~= array.size()/2) Analysis

The pivot is in the middle

$$T(N) = 2T(N/2) + cN, \quad N > 1$$

similar to mergesort analysis

$$T(N) = cN \log N + N = O(N \log N)$$

Average-Case Analysis

- Each of the sizes of S1 is equallt likely.
- The sizes are in range {0,…,N-1}
- The probability of an array having one of these sizes is: 1/N
- Assuming partitioning strategy is random
  - Otherwise analysis is not correct!
- The the *average* vaue of T(i) is like the following:

$$T(i) = 1/N \sum_{j=0}^{N-1} T(j) = T(N-i-1)$$

Average-Case Analysis

$$T(N) = 2\frac{1}{N}\left(\sum_{j=0}^{N-1} T(j)\right) + cN$$ Equation 1

Multiply the above equation by N

$$NT(N) = 2\left(\sum_{j=0}^{N-1} T(j)\right) + cN^2$$ Equation 2

Substitute N with N - 1

$$(N-1)T(N-1) = 2\left(\sum_{j=0}^{N-2} T(j)\right) + c(N-1)^2$$ Equation 3

Average-Case Analysis

Subtract equation 3 from equation 2

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c$$

$$NT(N) = (N+1)T(N-1) + 2cN$$ (ignore c)

divide both sides with $N(N+1)$

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1}$$ E1

Average-Case Analysis

Now telescope (write down equtions depending on smaller N)

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2c}{N} \quad \text{E2}$$

$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2c}{N-1} \quad \text{E3}$$

....

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3} \quad \text{E4}$$

Average-Case Analysis

Add all these equations E1 through E4 and obtain:

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{N+1}\frac{1}{i}$$

The sum is:

$$\sum_{i=3}^{N+1}\frac{1}{i} = \log_e(N+1) + 0.577 - \frac{3}{2}$$

Then

$$\frac{T(N)}{N+1} = O(\log N)$$

So the result is

$$T(N) = O(N \log N)$$

External Sorting

- So far we have assumed that all the input data can fit into main memory (RAM)
  - This means random access to data is possible and is not very costly.
- Algorithms such as *shell-sort*, and *quick-sort* make random access to array elements.
- If data is in a *hard-disk* or in a *tape* (in a *file*) random access is very costly.
- *External sorting algorithms* deal with these cases and can sort very large input sizes.

External Sorting

- External sorting algorithms makes sequential accesses to a storage device.
  - Tape or hard-disk.
  - In this way, the setup cost of retrieval is got rid of.
- Our model for external devices are (tapes)
  - They will be read from and written to sequentially.
    - In forward or reverse direction.
  - We can rewind the head to the beginning of the device (tape)
  - Assume we have at least three tape drives.

The Simple Algorithm

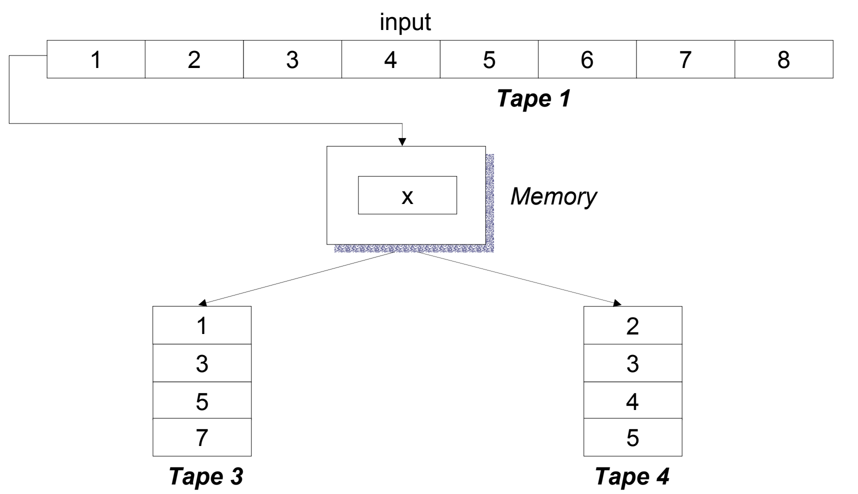
- Uses the merge idea from mergesort.
- Assume data is stored in a tape.
- Assume we have 4 tapes available.
- We will read M items at a time from input tape.
- We will sort them in memory and write to one of the output tapes. (set of M items will be called a ***Run***)
- We will continue doing this until we finish with the input.
- Then we will go to the merge step.

Algorithm Sketch

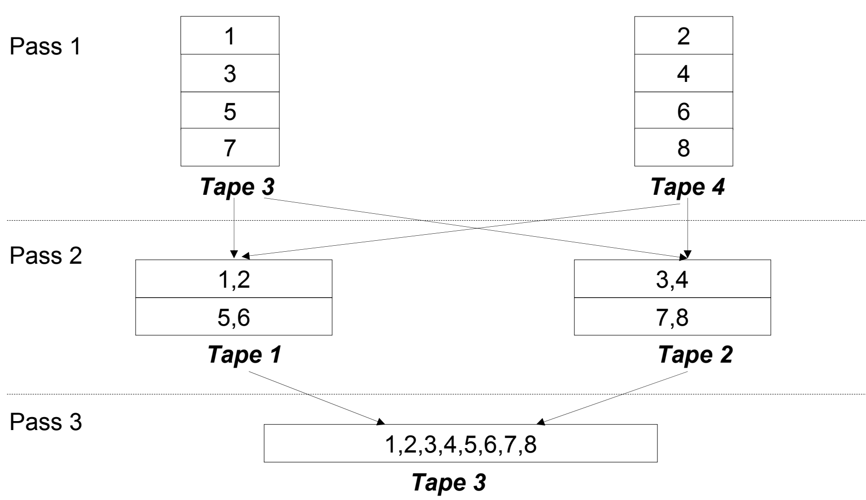
1. ***Constructing the runs***
   1. If tape 1 is not finished
      1. Read M items (if available) from tape1
      2. Sort them in memory
      3. Write them to tape 3 ***(these M items is called one run)***
   2. If tape 1 is not finished
      1. Read M items (if available) from tape 1
      2. Sort them in memory
      3. Write them to tape 4
   3. *Repeat steps 1 and 2 until tape 1 is finished.*
2. ***Merging runs***
   1. Merge runs in tapes 3 and 4 into tape 1 and 2.
      1. By taking *one run from tape 3* and *one run from tape 4*.
      2. *Continue in this way*

      At the end of this we have runs of size 2*M in tape 1 and 2
   2. Merge runs in tape 1 and 2 into tapes 3 and 4.
      At the end of this we have runs of size 4*M in tape 3 and 4.
   3. *Repeat steps 1 and 3 until we have a single run of size N (input size)*

Idea – constructing the runs



Idea – merging the runs



Example (M=3)

T1	81	94	11	96	12	35	17	99	28	58	41	75	15
T2													
T3													
T4													

After constructing the runs

T1													
T2													
T3	11	81	94	17	28	99	15						
T4	12	35	96	41	58	75							

After first pass

T1	11	12	35	81	94	96	15						
T2	17	28	41	58	75	99							
T3													
T4													

After second pass

T1													
T2													
T3	11	12	17	28	35	51	58	75	81	94	96	99	
T4	15												

After third pass

T1	11	12	15	17	28	35	51	58	75	81	94	96	99
T2													
T3													
T4													

Polyphase Merge

- In the previous example, we have used 4 tapes.
  - We did 2-way merge
  - It is possible to use 3 tapes in 2-way merge
- We can perform k-way merge similarly.
  - We need 2k tapes for simple algorithm
  - We need k+1 tapes for *polyphase* merge

Polyphase merge

- The idea is to not put the runs evenly to output tapes.
  - Some tapes should have more runs than the others.
- For two way merge
  - Have the number of runs in output tapes according to the *Fibonacci* numbers
    - Input = 8 ➔ output tape 1 = 3, output tape 2 = 5
    - Input = 13 ➔ output tape 1 = 5, output tape 2 = 8
    - Input = 21 ➔ output tape 1 = 13, output tape 2 = 8
    - …..
  - Add some dummy items to input if the size is not Fibonacci.

Assume N = 33 (input size)

	After Run Const.	After T3+T2	After T1+t2	After T1+T3	After T2+T3	After T1+T3	After T2+T3	After T2+T3
T1	0	13	5	0	3	1	0	1
T2	21	8	0	5	2	0	1	0
T3	13	0	8	3	0	2	1	0

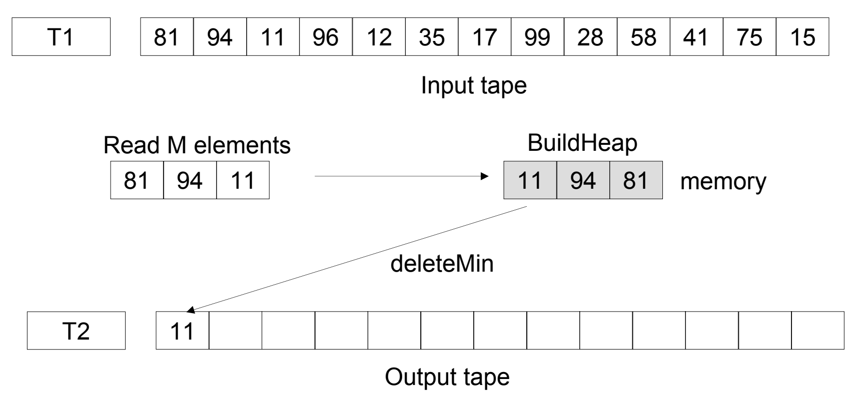
Run size

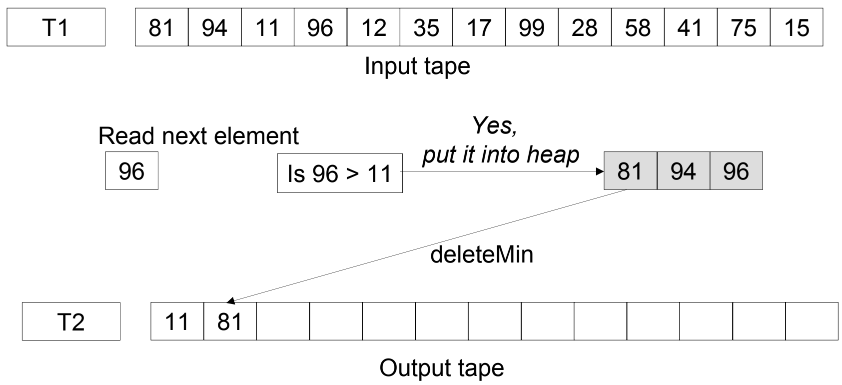
All run sizes are Fibonacci numbers.

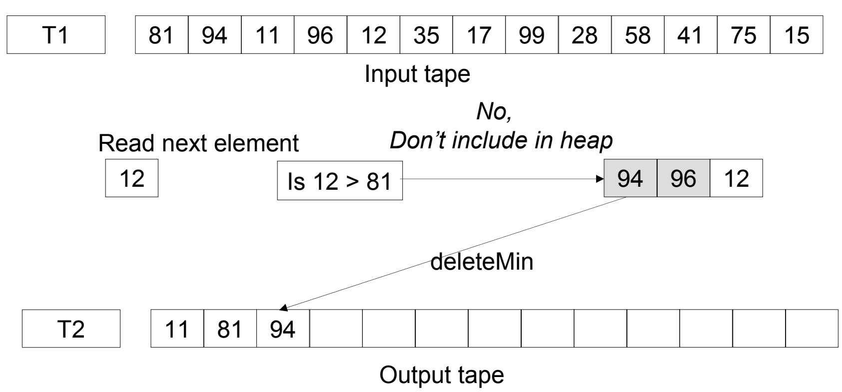
Replacement Selection

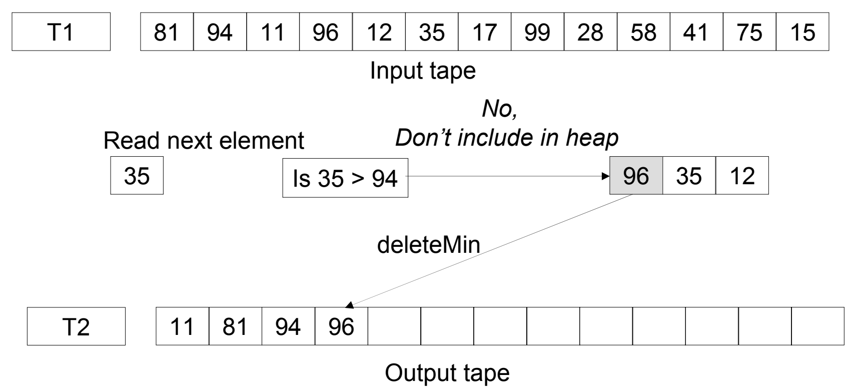
- A method for constructing the runs.
- Will produce variable sized runs.
  - All runs do not have equal sizes

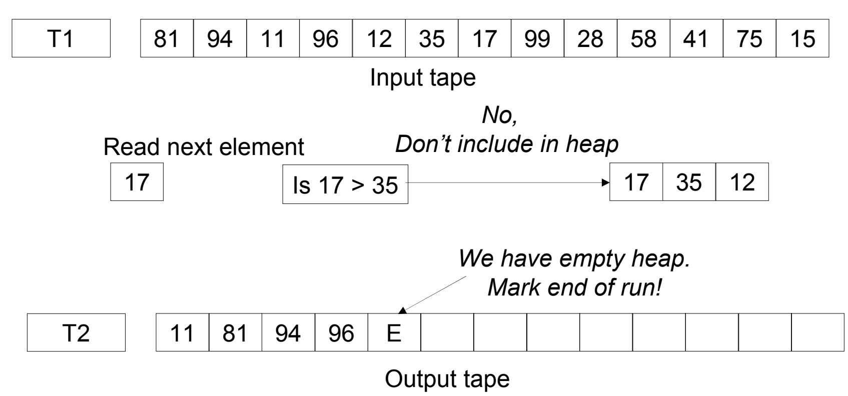
Example

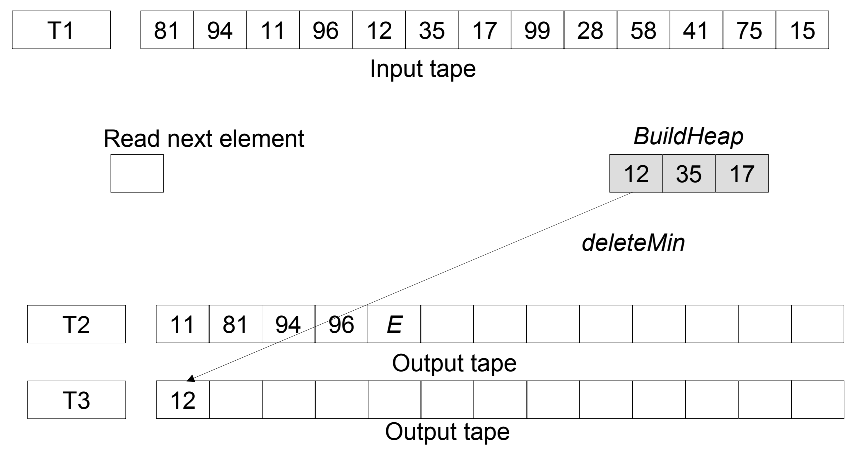


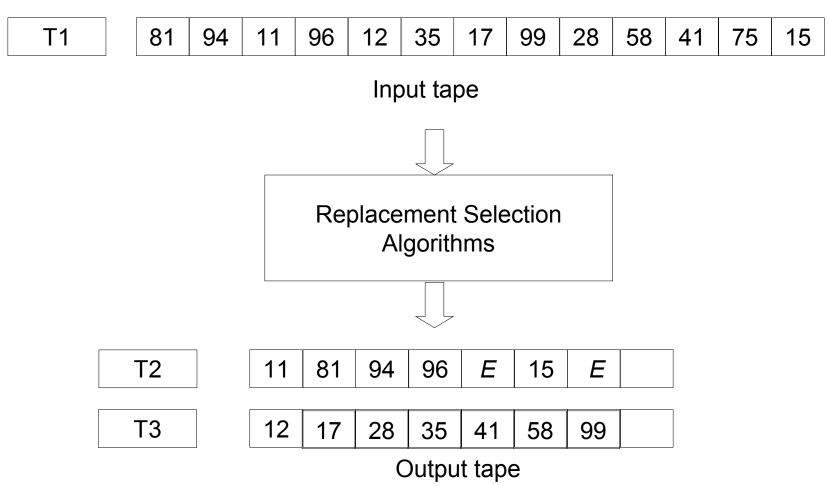

T1
81 94 11 96 12 35 17 99 28 58 41 75 15
Input tape
Read next element
96
Is 96 > 11
Yes,
put it into heap
81 94 96
deleteMin
T2
11 81
Output tape


T1
81 94 11 96 12 35 17 99 28 58 41 75 15
Input tape
Read next element
12
Is 12 > 81
No,
Don't include in heap
94 96 12
deleteMin
T2
11 81 94
Output tape


T1
81 94 11 96 12 35 17 99 28 58 41 75 15
Input tape
No,
Don't include in heap
Read next element
35
Is 35 > 94
96 35 12
deleteMin
T2
11 81 94 96
Output tape


T1
81 94 11 96 12 35 17 99 28 58 41 75 15
Input tape
No,
Don't include in heap
Read next element
17
Is 17 > 35
17 35 12
We have empty heap.
Mark end of run!
T2
11 81 94 96 E
Output tape


T1
81 94 11 96 12 35 17 99 28 58 41 75 15
Input tape
Read next element
BuildHeap
12 35 17
deleteMin
T2
11 81 94 96 E
Output tape
T3
12
Output tape


T1
81 94 11 96 12 35 17 99 28 58 41 75 15
Input tape
Replacement Selection
Algorithms
T2
11 81 94 96 E 15 E
T3
12 17 28 35 41 58 99
Output tape