Heapsort

Consider the following version of Selection Sort (sometimes called Max sort)

```
def maxSort(A,n):
    for k = n-1 downto 1
        x = max(A[0],A[1],...,A[k])
        A[k] ↔ x
```

A straightforward implementation requires $\Theta(n^2)$ time, because of the time spent repeatedly finding the maximum of the first k items:

$$\sum_{k=1}^{n-1} k = \Theta(n^2).$$

But we can speed this up by using a binary heap.

Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of elements.
  - Each element has an associated key, which corresponds to a priority.
  - Supports the following operations
    - Insert an element with a given priority
    - Delete an element
    - Select the element with highest priority currently in the priority queue.
  - Highest priority may correspond to the lowest key value or to the highest key value, depending on the application.

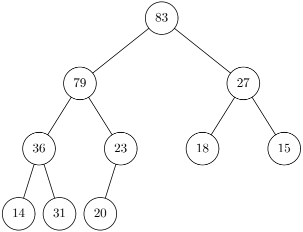
Binary Heaps

- Implementation of priority queue
- Elements are stored in an array.
- Conceptually, the corresponds to a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
- The next few slides assume a max-heap.
- Heap invariant: For any element v other than the root,

$$\mathtt{key}\,(\mathtt{parent}(v)) \geq \mathtt{key}(v)$$

Navigating the binary tree

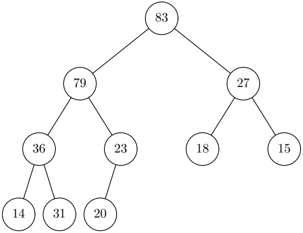
- Left son of $H[i]$ is $H[2i+1]$ (provided $2i+1 < n$, where $n = H.\text{size}$)
- Right son of $H[i]$ is $H[2i+2]$ (provided $2i+2 < n$)
- Parent of $H[i]$ is $H[(i-1)/2]$ (provided $i > 0$)



83	79	27	36	23	18	15	14	31	20
0	1	2	3	4	5	6	7	8	9

Heap operations in a max-heap:

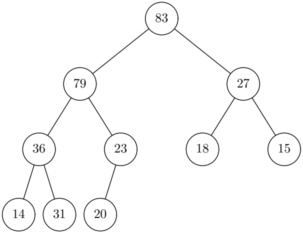
- `FindMax(H)`: Find maximum element in the heap
- `ExtractMax(H)`: Find maximum element and delete it from the heap
- `Insert(H,x)`: Insert the new element x in the heap
- `Delete(H,i)`: Delete the element at location i from the heap



FindMax: Find maximum element in the heap

Findmax is easy: just report the value at the root.

```
def FindMax(H):
    return H[0]
```



Helper functions

The other operations require some data movement. The heap invariant must be preserved after each operation. We define two helper functions.

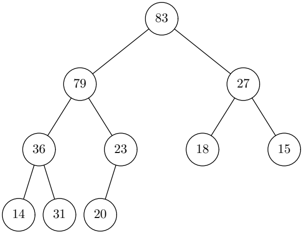
- `SiftUp(H,i)`: Move the element at location i up to its correct position by repeatedly swapping the element with its parent, as necessary.
- `SiftDown(H,i)`: Move the element at location i down to its correct position by repeatedly swapping the element with the child having the larger key, as necessary.

[GT] calls these "up-heap bubbling" and "down-heap bubbling"

SiftUp: Sift an element up to its correct position

```
def SiftUp(H,i):
    parent = (i-1)/2;
    if (i > 0) and (H[parent].key < H[i].key):
        H[i] ↔ H[parent]
        SiftUp(H,parent)
```

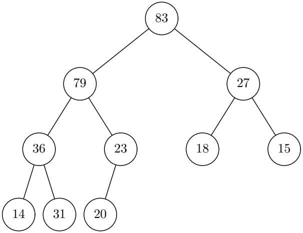
Work: at most 1 comparison at each level, so total work $\in O(\log n)$



SiftDown: Sift an element down to its correct position

```
def SiftDown(H,i):
    n = H.size // number of elements in heap
    left = 2i+1; right = 2i+2
    if (right < n) and (H[right].key > H[left].key)
        largerChild = right
    else largerChild = left
    if (largerchild < n) and (H[i].key < H[largerChild].key)
        H[i] ↔ H[largerchild]
        SiftDown(H,largerchild)
```

Work: at most 2 comparison at each level, so total work $\in O(\log n)$

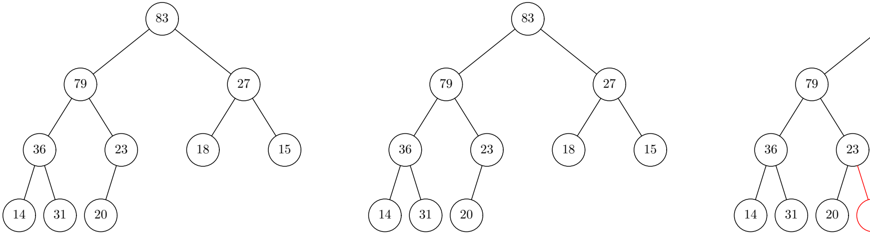


Insert: Insert the new element x

```
def Insert(H,x):
    H.size = H.size+1 // increment number of elements
    k = H.size-1 //index of last position
    H[k] = x //insert x in last position
    SiftUp(H,k)
```

Cost$= O(\log n)$

`Insert(H,81)`

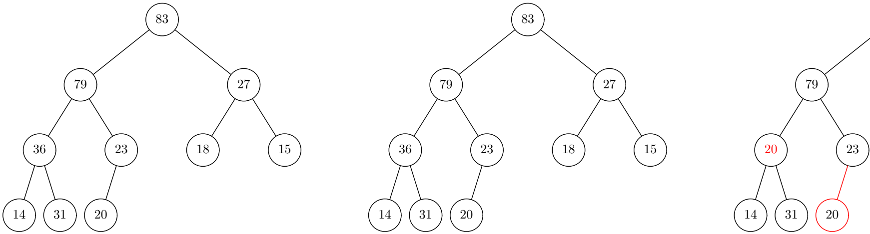


Delete: Delete the element at location i

```
def Delete(H,i):
    k = H.size-1 //index of last position
    H[i] = H[k] // overwrite element being deleted with
                   element in last position
    H.size = H.size-1 // decrement number of elements
    SiftUp(H,i) // either SiftUp or SiftDown will do nothing
    SiftDown(H,i)
```

Cost$=O(\log n)$

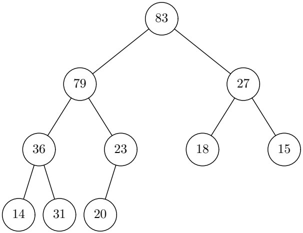
`Delete(H,3)`



ExtractMax: Find maximum element and delete it

```
def ExtractMax(H):
    x = H[0]
    Delete(H,0)
    return x
```

Cost: $O(\log n)$



Constructing a heap

How do we efficiently construct a brand-new heap storing n given elements?

If we insert the elements one at a time, time spent on kth insertion is $O(\log k)$. So total time is

$$O\left(\sum_{k=1}^{n-1} \log k\right) = O\left(n \log n\right)$$

There is a better way that only requires $O(n)$ time...

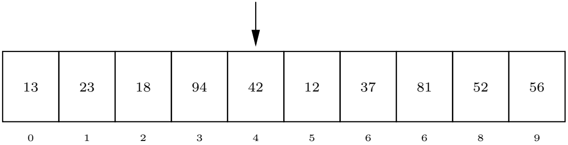
Constructing a heap in $O(n)$ time

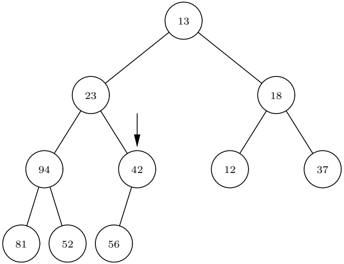
1. Put the data in H, in arbitrary order. (So H stores the correct data, but does not satisfy the heap invariant.)
2. Run the following `Heapify` function.

```
def heapify(H,n)
    for i := ⌊(n-1)/2⌋ down to 0:
        SiftDown(H,i)
```

Heapify example

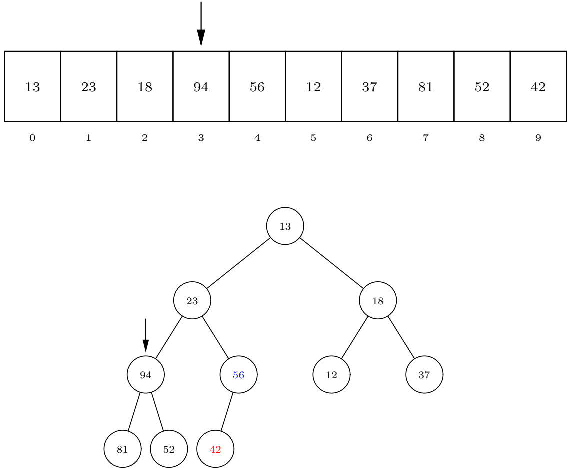
13 23 18 94 42 12 37 81 52 56





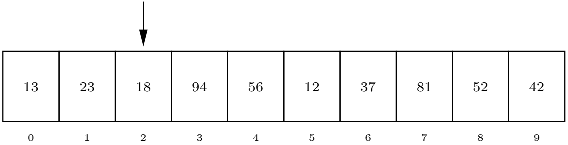
Heapify example, continued

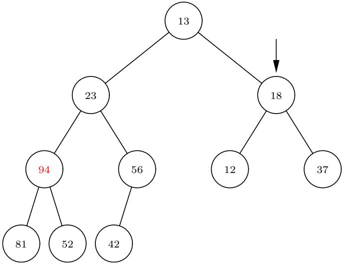
13 23 18 94 42 12 37 81 52 56



Heapify example, continued

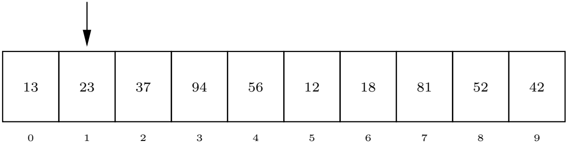
13 23 18 94 42 12 37 81 52 56

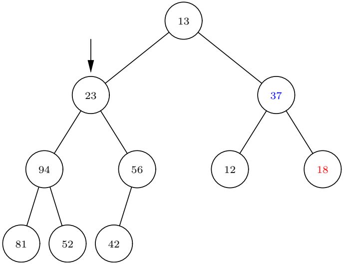




Heapify example, continued

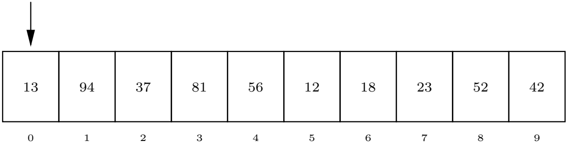
13 23 18 94 42 12 37 81 52 56

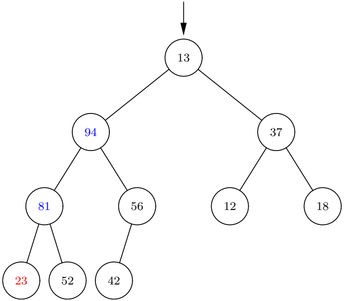




Heapify example, continued

13 23 18 94 42 12 37 81 52 56

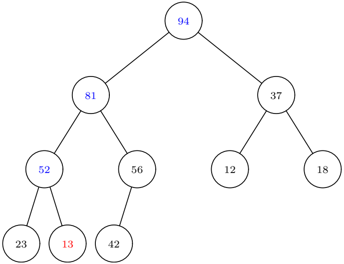




Heapify example, continued

13 23 18 94 42 12 37 81 52 56

94	81	37	52	56	12	18	23	13	42
0	1	2	3	4	5	6	7	9	9



Analysis of heap construction algorithm using Heapify

```
Algorithm heapify(H,n);
   for i := ⌊(n-1)/2⌋ down to 0 do
      SiftDown(H,i);
```

- Correctness: After SiftDown(H,i) is executed, subtree rooted at node i satisfies heap invariant. (Can show by induction).
- Running time: `Heapify` runs in $O(n)$ time. We will prove this two diffent ways.
  1. An algebraic proof
  2. An amortization (banker's) proof

Analysis # 1: Algebraic proof

- Suppose the tree has n nodes and d levels (so $2^d \leq n < 2^{d+1}$).
- If node i is at level j, `SiftDown(H,i)` needs $\leq 2(d-j)$ comparisons.
- There are at most 2^j nodes at level j.
- So total number of comparisons is no more than:

$$\begin{aligned}
\sum_{j=0}^{d} 2(d-j)2^j &= 2d\sum_{j=0}^{d} 2^j - 2\sum_{j=0}^{d} j2^j \\
&= 2d(2^{d+1}-1) - 2\left[(d-1)2^{d+1}+2\right] \\
&= 2d2^{d+1} - 2d - 2d2^{d+1} + 2\cdot 2^{d+1} - 4 \\
&= 4\cdot 2^d - 2d - 4 \\
&\leq 4n = O(n)
\end{aligned}$$

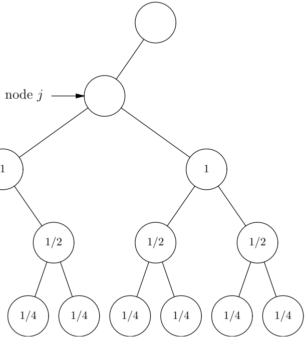
So heap can be constructed using $O(n)$ comparisons.

Analysis # 2: Amortized analysis (banker's argument)

- We think of each comparison as costing 1 dollar.
- Show that comparison costs can be "charged" to nodes so that:
  - For every comparison, some combination of nodes collectively gets charged $1
  - No node gets charged more than $2 total.
- This proves that there are no more than $2n$ comparisons.

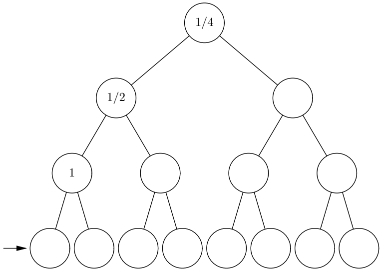
The charging scheme

- When we perform the operation `SiftDown(H,i)` at most 2 comparisons per level are performed at each level below *i*'s level. The comparisons are always against descendants of node *i*.
- Charge the descendants as follows:



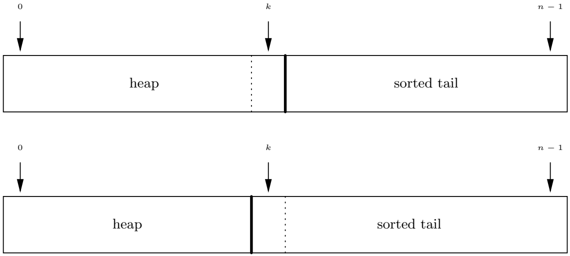
Adding up the charges

- How much does each node get charged over the entire `Heapify` operation? No more than
  - 1 from parent
  - 1/2 from grandparent
  - 1/4 from great-grandparent
  - etc.
- Hence total charge to each node $< 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 2$
- So `Heapify` runs in $O(n)$ time, QED.



Heapsort: in-place version

```
def heapsort(A,n);
    heapify(A,n); // form max heap using array A
    for k = n-1 down to 1 do
        A[k] := ExtractMax(A);
```



Analysis of Heapsort

- Storage: $O(1)$ extra space (hence in place)
- Time:
  - `Heapify`: $O(n)$
  - All calls to `ExtractMax`:

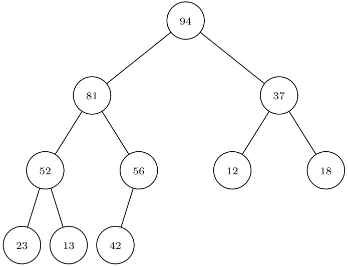
$$\sum_{k=1}^{n-1} O\left(\log(k+1)\right) = O(n \log n)$$

  - Hence total time is $O(n \log n)$.

Heapsort example

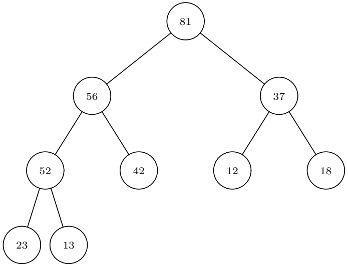
Sort: 13 23 18 94 42 12 37 81 52 56

Heapify:



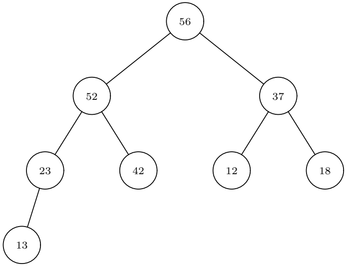
0	1	2	3	4	5	6	7	8	9
94	81	37	52	56	12	18	23	13	42

Heapsort example, continued



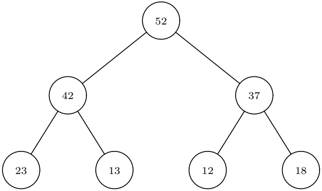
81	56	37	52	42	12	18	23	13	94
0	1	2	3	4	5	6	7	8	9

Heapsort example, continued



56	52	37	23	42	12	18	13	81	94
0	1	2	3	4	5	6	7	8	9

Heapsort example, continued



52	42	37	23	13	12	18	56	81	94
0	1	2	3	4	5	6	7	8	9

Exercise: Finish this example.

Heapsort: Alternate version

- Uses a min-heap (instead of a max-heap)
- Outputs items in sorted order rather than storing them back in the array

```
def heapsort(A,n):

    heapify(A,n) // Form min heap
    for k := 1 to n:
        x := ExtractMin(A)
        output(x)
```

- Same analysis as previous version: $O(n \log n)$ time, $O(1)$ extra space
- If we stop after computing the first k entries, total work is

$$O(n + k \log n)$$

Comparison-based sorts: Summary/Comparison

Sort	Worst-case Time	Storage Requirement	Remarks
Insertion Sort	$\Theta(n^2)$	In-place	Good if input is almost sorted.
QuickSort	$\Theta(n^2)$	$O(\log n)$ extra for stack	$O(n \log n)$ expected time.
Mergesort	$\Theta(n \log n)$	$O(n)$ extra for merge	
Heapsort	$\Theta(n \log n)$	In-place	Can output k smallest in sorted order in $O(n + k \log n)$ time.

Stable sorting

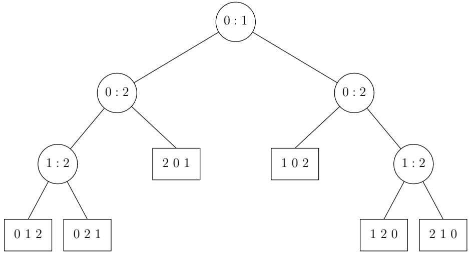
A sort is stable if keys having the same value appear in the same order in the output array as they do in the input array.

Sort	Stable (without special care)?
Insertion Sort	Yes
Quick-Sort	No
Merge-Sort	Yes (as described here)
Heap-Sort	No

Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size n using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $1, \ldots n$.

Example: Decision tree for sorting 3 elements



Exact lower bound on comparison-based sorting

1. Any algorithm for sorting a list of size n can be modeled by a decision tree with at least $n!$ leaf nodes.
2. The worst-case number of comparisons for the algorithm is the depth of the decision tree. (Remember, root has depth 0).
3. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.

Hence any algorithm for sorting a list of size n using only comparisons must perform at least $\lceil \lg n! \rceil$ comparisons in the worst case.

Asymptotic lower bound on comparison-based sorting

- As we just proved, any algorithm for sorting a list of size n using only comparisons must perform at least $\lceil \lg n! \rceil$ comparisons in the worst case.
- $\lceil \lg n! \rceil = \Theta(n \log n)$. Hence any algorithm for sorting a list of size n using only comparisons must perform at least $\Theta(n \log n)$ comparisons in the worst case.
- Conclusions:
  1. Heapsort and Mergesort are asymptotically optimal.
  2. The lower bound is asymptotically tight (i.e., cannot be improved asymptotically)

Comparisons by Mergesort vs. the exact lower bound

- Sorting lower bound: $\lceil \lg n! \rceil$.
- Mergesort: Solution of

$$W(n) = \begin{cases} n - 1 + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + W\left(\left\lfloor \frac{n}{2} \right\rfloor\right), & n > 1 \\ 0, & n = 1 \end{cases}$$

- Comparison:

n	Lower Bound	Merge Sort		n	Lower Bound	Merge Sort
1	0	0		10	22	25
2	1	1		11	26	29
3	3	3		12	29	33
4	5	5		13	33	37
5	7	8 $\leftarrow$		14	37	41
6	10	11		15	41	45
7	13	14		16	45	49
8	16	17		17	49	54
9	19	21		18	53	59

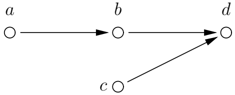
Optimally sorting 5 elements

- According to table on previous slide:
  - Mergesort requires 8 comparisons to sort 5 elements
  - The lower bound says we need at least 7 comparisons to sort 5 elements
- Question: Is it possible to sort 5 elements using only 7 comparisons?
- Answer: Yes

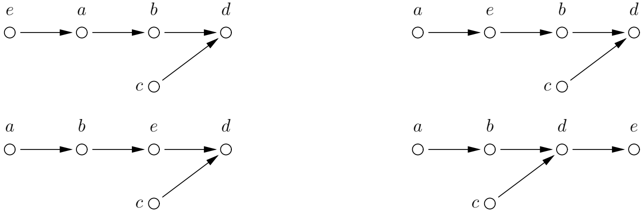
Call the 5 elements `a`, `b`, `c`, `d`, `e`...

Sorting 5 elements with 7 comparisons

`if` $a > b$: $a \leftrightarrow b$
`if` $c > d$: $c \leftrightarrow d$
`if` $b > d$: $b \leftrightarrow d$, $a \leftrightarrow c$ (3 comparisons)



Find position for e in [a,b,d] (2 comparisons)



Find position for c (4 cases, 2 comparisons in each case)

Address-Calculation Sorting Algorithms

- Based on data values.
- Performance is not limited by $\Omega(n \log n)$ bound, but does depend on data values.
- Comparisons are not necessarily a reasonable measure of work performed.
- We will discuss 3 algorithms:
  1. Counting sort
  2. Bucket sort
  3. Radix sort

Counting sort

Underlying idea: Suppose there are are exactly j elements $\leq x$

- If x only appears once, then it belongs in the jth location.
- If x appears more than once and we want a stable sort:
  - Last occurrence of x belongs in jth location
  - Next-to-last occurrence of x belongs in $(j-1)$st location
  - etc.

Counting sort

- Assume:
  - We are sorting an array $A[1..n]$ of integers
  - Each integer is in the range $1..k$
  - Output array is $B[1..n]$
- Use an auxiliary array `locator[]`
  - `locator`$[x]$ contains the index of the position in the output array B where a value of x should be stored.
  - Initially, `locator`$[x]$ contains the number of elements $\leq x$
- Process input array from right to left
- When a value of x is encountered:
  - Copy it into location `locator`$[x]$ in the output array (i.e., into $B[$`locator`$[x]])$
  - Decrement `locator`$[x]$

Code for Counting sort

```
def CountingSort(A, B, n , k)
    //Initialize:  set each locator[x] to number of entries ≤ x
    for x = 1 to k do locator[x] = 0;
    for i = 1 to n do locator[A[i]] := locator[A[i]] + 1;
    for x = 2 to k do
        locator[x] = locator[x] + locator[x-1];
    //Fill output array, updating locator values
    for i := n down to 1 do
        B[locator[A[i]]] := A[i];
        locator[A[i]] := locator[A[i]] - 1;
```

Counting Sort Example

A:

1	2	3	4	5	6	7	8	9	10	1	2
1	3	5	7	5	7	3	8	7	4	1	3

`locator`:

1	2	3	4	5	6	7	8	1	2	3	4
1	1	3	4	6	6	9	10	1	1	3	4

B:

1	2	3	4	5	6	7	8	9	10	1	2

Bucket Sort

- Divide space of possible keys into contiguous subranges, or buckets.
- Three steps:
  1. Distribute keys into buckets
  2. Sort keys in each bucket
  3. Combine buckets.
- Simplest approach is to divide the space of possible keys into equal sized buckets.
- Typically use insertion sort in step 2.

Bucket Sort Example

Sort the following keys in the range 1-1000, using 10 equal-size buckets:

661 74 835 140 198 923 113 642 467 449

1. Distribute

1:	74
2:	140 198 113
3:	
4:	
5:	467 449
6:	
7:	661 642
8:	
9:	835
10:	923

2. Sort

1:	74
2:	113 140 198
3:	
4:	
5:	449 467
6:	
7:	642 661
8:	
9:	835
10:	923

3. Combine

74
113
140
198
449
467
642
661
835
923

Analysis of Bucket Sort

$$
\begin{aligned}
n &= \text{number of items to sort} \\
b &= \text{number of buckets} \\
s_i &= \text{number of items in bucket } i \ (i = 1, \ldots, b)
\end{aligned}
$$

Phase	Work
1. Distribution	$O(n)$
2. Sorting each bucket	$O(b + \sum_i s_i^2)$
3. Combining buckets	$O(b)$

Total work performed is:

$$O\left(n + \sum_{i=1}^{b} s_i^2 + b\right)$$

Storage is $O(n + b)$.

Special case of Bucket Sort

If the keys are distributed independently and uniformly over the buckets, and if $b = n$, then it can be shown that the expected total cost of the intra-bucket sorts is $O(n)$. The expected total work is then $O(n)$.

Another special case of Bucket Sort

- Suppose we have n integers in the range $1..b$ (or $0..b-1$).
- Use b buckets.
  1. Distribution Phase takes $O(n)$ time
  2. Sorting each bucket takes no time (!)
  3. Combining buckets takes $O(b)$ time

  Hence the total work performed is $O(n+b)$ (worst-case).

Example: Sort the vertices of a graph according to their degrees. Each vertex satisfies

$$0 \leq \text{degree}(v) \leq n-1,$$

so the sorting can be done in $O(n)$ time, $O(n)$ space.

(The same time bound can be achieved by using Counting Sort.)

Radix Sort

- Useful for sorting multi-field keys (e.g., dates)
- Also for multi-digit numbers (treat each digit as a field)
- Mimics old card-sorting machines
- Slightly counterintuitive because sorted on least-significant portion first

```
algorithm radix_sort(A,n);
   for field ranging from rightmost (least significant)
         to leftmost (most significant) do
      sort L on field using a stable sort
```

Recall: A sorting algorithm is stable if whenever two keys are equal, the algorithm preserves their order (i.e., does not reverse them.)

Radix Sort Example:

Sort the following numbers using radix sort (each digit is a field)
661 74 835 140 198 923 113 642 467 449

661		140		113		074
074		661		923		113
835		642		835		140
140		923		140		198
198		113		642		449
923	⇒	074	⇒	449	⇒	467
113		835		661		642
642		467		467		661
467		198		074		835
449		449		198		923

Note the importance of stability.

External Sorting

- Problem: Sorting a large file, bigger than available memory
- Assume
  1. n records in file
  2. m records can fit in memory at once ($m \ll n$)
  3. f input files can be open at once.

Polyphase Merge

- Phase 1:
  - Read in groups of m records
  - Sort each group
  - Write each run (sorted group) to a separate output file
- Subsequent phases: Repeatedly
  - Choose f files
  - Merge the contents of the f input files into a new output file
  - Delete the f input files
- For effiency, choose the smallest length files or use FIFO ordering

Polyphase Merge example ($n = 54$, $m = 4$, $f = 3$)

```
145 507 354 590 875  29   9 481  47 212 208 929 902 124 250  11
386 281 680 109 100 542  64 508 654 793 538 322 299 686 104 989
465 777 991 931 677 176 230 214 369 106 218 724 779 565 559 873
696 726 326 415 761 915
```

```
Phase 1:  145 507 354 590 ⇒  145 354 507 590 (Run 1)
          875  29   9 481 ⇒    9  29 481 875 (Run 2)
           47 212 208 929 ⇒   47 208 212 929 (Run 3)
          902 124 250  11 ⇒   11 124 250 902 (Run 4)
          386 281 680 109 ⇒  109 281 386 680 (Run 5)
          100 542  64 508 ⇒   64 100 508 542 (Run 6)
          654 793 538 322 ⇒  322 538 654 793 (Run 7)
          299 686 104 989 ⇒  104 299 686 989 (Run 8)
          465 777 991 931 ⇒  465 777 931 991 (Run 9)
          677 176 230 214 ⇒  176 214 230 677 (Run 10)
          369 106 218 724 ⇒  106 218 369 724 (Run 11)
          779 565 559 873 ⇒  559 565 779 873 (Run 12)
          696 726 326 415 ⇒  326 415 696 726 (Run 13)
          761 915         ⇒  761 915         (Run 14)
```

Polyphase Merge example, continued (subsequent phases)

(Run 1 + Run 2 + Run 3) ⇒ Run 15:

9 29 47 145 208 212 354 481 507 590 875 929

(Run 4 + Run 5 + Run 6) ⇒ Run 16:

11 64 100 109 124 250 281 386 508 542 680 902

(Run 7 + Run 8 + Run 9) ⇒ Run 17:

104 299 322 465 538 654 686 777 793 931 989 991

(Run 10 + Run 11 + Run 12) ⇒ Run 18:

106 176 214 218 230 369 559 565 677 724 779 873

(Run 13 + Run 14 + Run 15) ⇒ Run 19:

9 29 47 145 208 212 326 354 415 481 507 590 696 726 761 875
915 929

(Run 16 + Run 17 + Run 18) ⇒ Run 20:

11 64 100 104 106 109 124 176 214 218 230 250 281 299 322 369
386 465 508 538 542 559 565 654 677 680 686 724 777 779 793 873
902 931 989 991

(Run 19 + Run 20) ⇒ Run 21:

9 11 29 47 64 100 104 106 109 124 145 176 208 212 214 218
230 250 281 299 322 326 354 369 386 415 465 481 507 508 538 542
559 565 590 654 677 680 686 696 724 726 761 777 779 793 873 875
902 915 929 931 989 991

Replacement Selection

The initial runs can be made longer by using an improvement called Replacement Selection. When a key is written, the next key is read.

- If the new key is $\geq$ the last key written, it is made part of the current run.
- If the new key is $<$ the last key written, it is saved for the next run.

Replacement Selection Example

145 507 354 590 875 29 9 481 47 212 208 929 902 124 250 11 386 281 680
109 100 542 64 508 654 793 538 322 299 686 104 989 465 777 991 931 677
176 230 214 369 106 218 724 779 565 559 873 696 726 326 415 761 915

Memory				Run	Run Contents
145_1	507_1	354_1	590_1	1	
875_1	507_1	354_1	590_1	1	145
875_1	507_1	29_2	590_1	1	145 354
875_1	9_2	29_2	590_1	1	145 354 507
875_1	9_2	29_2	481_2	1	145 354 507 590
47_2	9_2	29_2	481_2	1	145 354 507 590 875

Replacement Selection Example, continued

145 507 354 590 875 29 9 481 47 | 212 208 929 902 124 250 11 386 281 680
109 100 542 64 508 654 793 538 322 299 686 104 989 465 777 991 931 677
176 230 214 369 106 218 724 779 565 559 873 696 726 326 415 761 915

Memory				Run	Run Contents
47_2	9_2	29_2	481_2	2	
47_2	212_2	29_2	481_2	2	9
47_2	212_2	208_2	481_2	2	9 29
929_2	212_2	208_2	481_2	2	9 29 47
929_2	212_2	902_2	481_2	2	9 29 47 208
929_2	124_3	902_2	481_2	2	9 29 47 208 212
929_2	124_3	902_2	250_3	2	9 29 47 208 212 481
929_2	124_3	11_3	250_3	2	9 29 47 208 212 481 902
386_3	124_3	11_3	250_3	2	9 29 47 208 212 481 902 929

Replacement Selection Example, continued

145 507 354 590 875 29 9 481 47 212 208 929 902 124 250 11 386 | 281 680
109 100 542 64 508 654 793 538 322 299 686 104 989 465 777 991 931 677
176 230 214 369 106 218 724 779 565 559 873 696 726 326 415 761 915

Memory				Run	Run Contents
386_3	124_3	11_3	250_3	3	
386_3	124_3	281_3	250_3	3	11
386_3	680_3	281_3	250_3	3	11 124
386_3	680_3	281_3	109_4	3	11 124 250
386_3	680_3	100_4	109_4	3	11 124 250 281
542_3	680_3	100_4	109_4	3	11 124 250 281 386
64_4	508_4	100_4	109_4	3	11 124 250 281 386 542 680

With Replacement Selection: 7 initial runs, 10 total runs (vs. 14 and 21)

Run 1:
145 354 507 590 875

Run 2:
9 29 47 208 212 481 902 929

Run 3:
11 124 250 281 386 542 680

Run 4:
64 100 109 508 538 654 686 793 989

Run 5:
104 299 322 465 677 777 931 991

Run 6:
176 214 218 230 369 565 724 779 873

Run 7:
106 326 415 559 696 726 761 915

Subsequent runs

(Run 1 + Run 2 + Run 3) $\Rightarrow$ Run 8:

9 11 29 47 124 145 208 212 250 281 354 386
481 507 542 590 680 875 902 929

(Run 4 + Run 5 + Run 6) $\Rightarrow$ Run 9:

64 100 104 109 176 214 218 230 299 322 369 465
508 538 565 654 779 793 873 931
989 991

(Run 7 + Run 8 + Run 9) $\Rightarrow$ Run 10:

9 11 29 47 64 100 104 106 109 124 145 176
208 212 214 218 230 250 281 299 322 326 354 369
386 415 465 481 507 508 538 542 559 565 590 654
677 680 686 696 724 726 761 777 779 793 873 875
902 915 929 931 989 991

Polyphase Merge, Replacement Selection Summary

- On the average, replacement selection doubles the sizes of the runs, assuming uniform distribution of the sort keys.
- $m = 4$ and $f = 3$ are for purposes of illustration only. Realistic values are much larger.
- Use a min-heap while building initial runs
- Use a min-heap during the merging phase
- Additional complications enter when performing tape-to-tape sorts (limited number of tape drives).
- Encyclopedic reference: Donald Knuth, *Sorting and Searching*, The Art of Computer Programming, Vol. 3