

Chap 7. Sorting (2)

Contents

1. Motivation
2. Insertion Sort, Shell Sort
3. Quick Sort
4. How Fast Can We Sort?
5. Merge Sort
6. Heap Sort
7. Sorting on Several Keys
9. Summary of Internal Sorting

7.6 Heap Sort

- Heap sort
 - Utilizes the *min heap* structure
 - Requires only *a fixed* amount of additional storage
 - Worst/average case time complexity : $O(n \log n)$

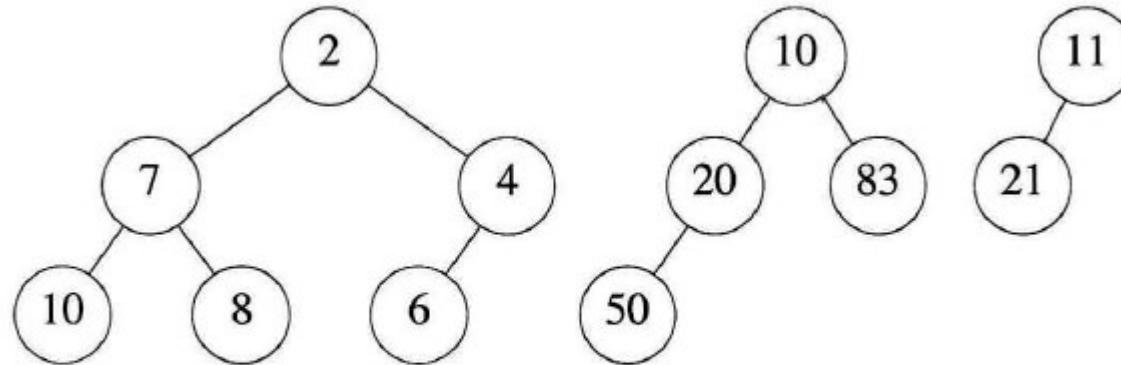
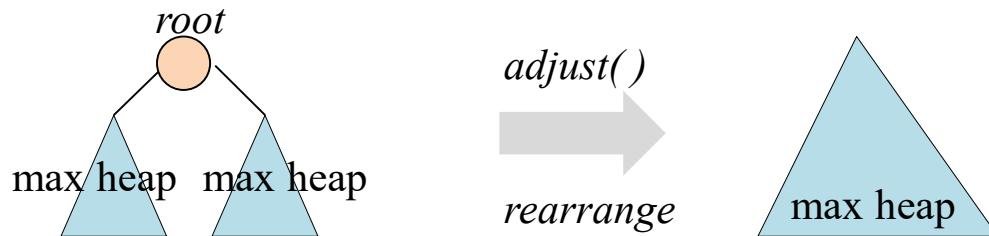


Figure 5.26: Min heaps

7.6 Heap Sort

- Creating the max heap of n records by using function *adjust*
 - faster than by inserting introduced in Chapter 05



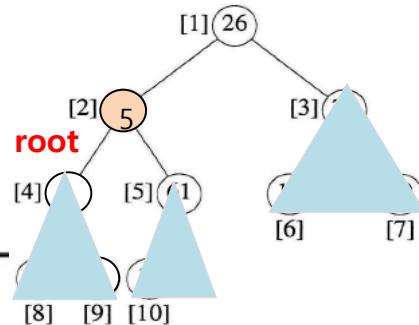
```
void heapSort(element a[], int n)
/* perform a heap sort on a[1:n] */
int i, j;
element temp;

1. for (i = n/2; i > 0; i--)
2.   adjust(a,i,n);
   for (i = n-1; i > 0; i--) {
     SWAP(a[1],a[i+1],temp);
     adjust(a,1,i);
   }
}
```

Program 7.13: Heap sort

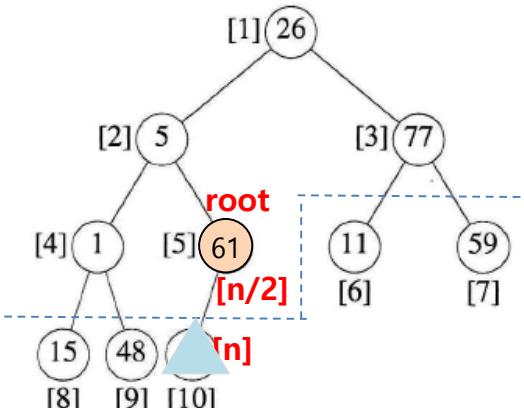
1. Create *an initial max heap* by using *adjust* repeatedly.
2. Repeat the following pass $n-1$ times to *sort* an array $a[1:n]$.
 - ① Swap the *first* and *last* records in the heap
 - ② Decrease the heap size and *readjust* the heap

```
void adjust(element a[], int root, int n)
{/* adjust the binary tree to establish the heap */
    int child,rootkey;
    element temp;
    temp = a[root];
    rootkey = a[root].key;
    child = 2 * root;                                /* left child */
    while (child <= n) {
        if ((child < n) &&
            (a[child].key < a[child+1].key))
            child++;
        if (rootkey > a[child].key) /* compare root and
                                       max. child */
            break;
        else {
            a[child / 2] = a[child]; /* move to parent */
            child *= 2;
        }
    }
    a[child/2] = temp;
}
```

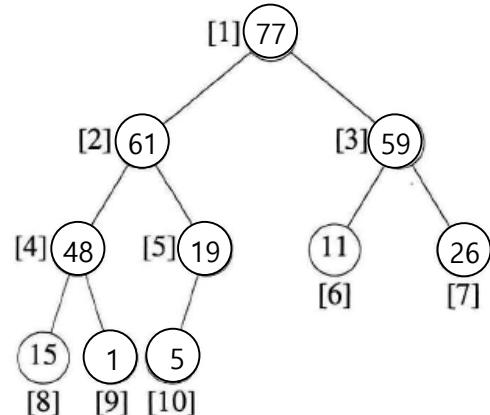
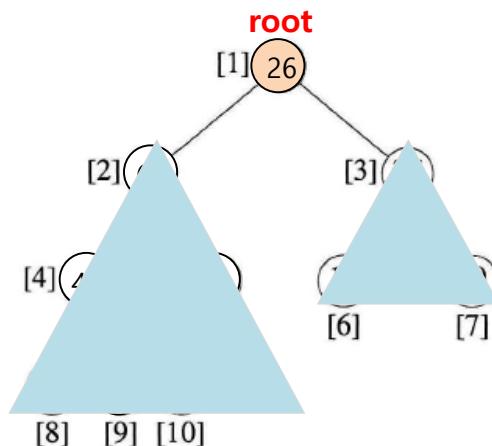
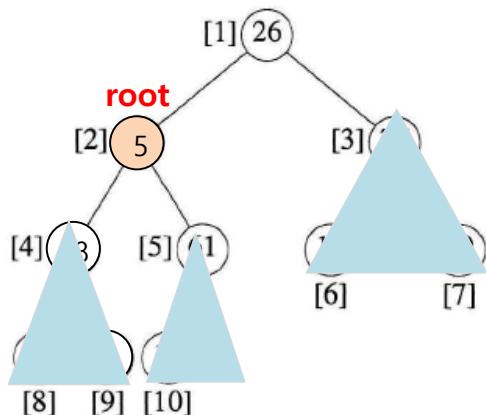
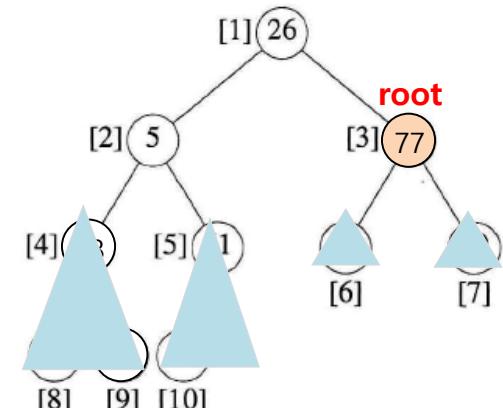
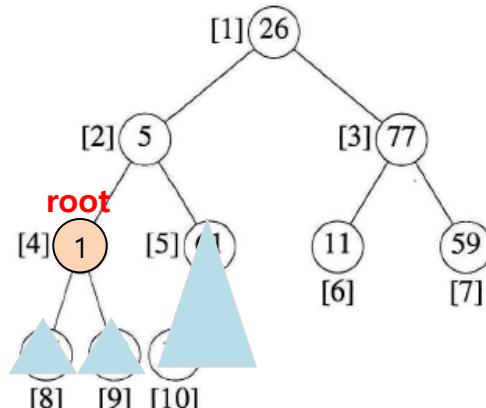


Program 7.12: Adjusting a max heap

1. Creating an initial *max heap*

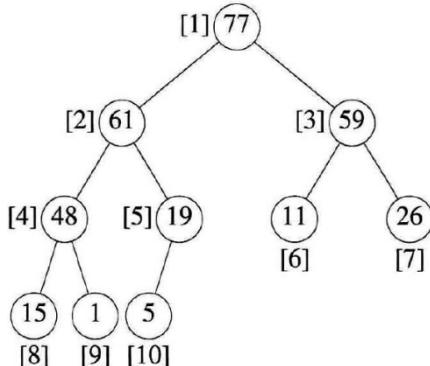


Input array

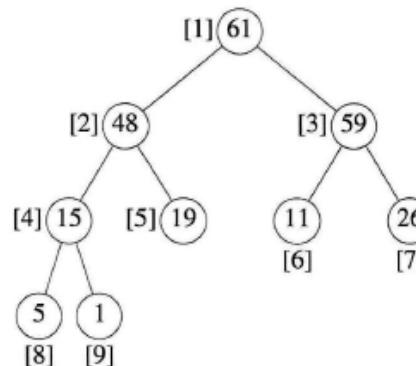


Initial heap

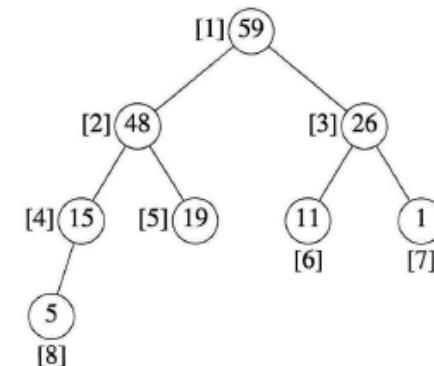
2. Sorting the array $a[1:n]$



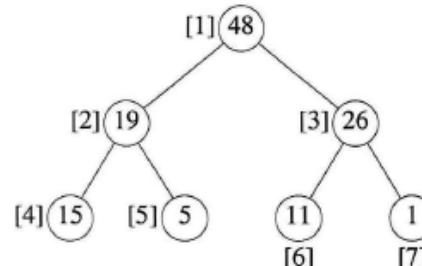
Initial heap



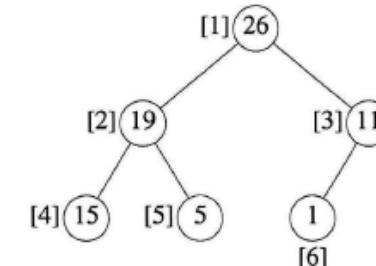
(a) Heap size = 9
Sorted = [77]



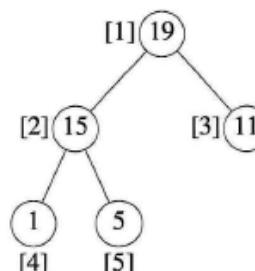
(b) Heap size = 8
Sorted = [61, 77]



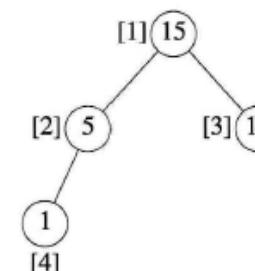
(c) Heap size = 7
Sorted = [59, 61, 77]



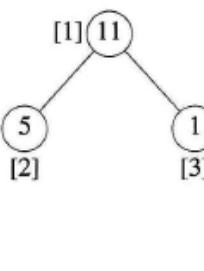
(d) Heap size = 6
Sorted = [48, 59, 61, 77]



(e) Heap size = 5
[26, 48, 59, 61, 77]



(f) Heap size = 4
[19, 26, 48, 59, 61, 77]



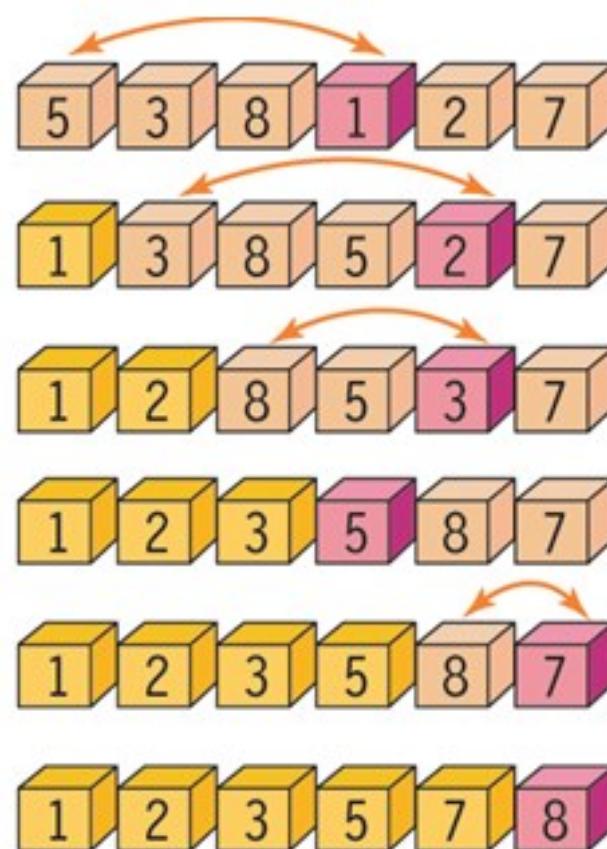
(g) Heap size = 3
[15, 19, 26, 48, 59, 61, 77]

Figure 7.8: Heap sort example

- **Analysis of *heapSort*:**
 - average case : $O(n \cdot \log_2 n)$
 - function *adjust* : $O(d)$, where d : depth of tree

selection sort

- A relatively easy to understand algorithm
- Sorts an array in *passes*
 - Each pass selects the next smallest element
 - At the end of the pass, places it where it belongs
- Performs:
 - $O(n^2)$ comparisons
 - $O(n)$ exchanges (swaps)

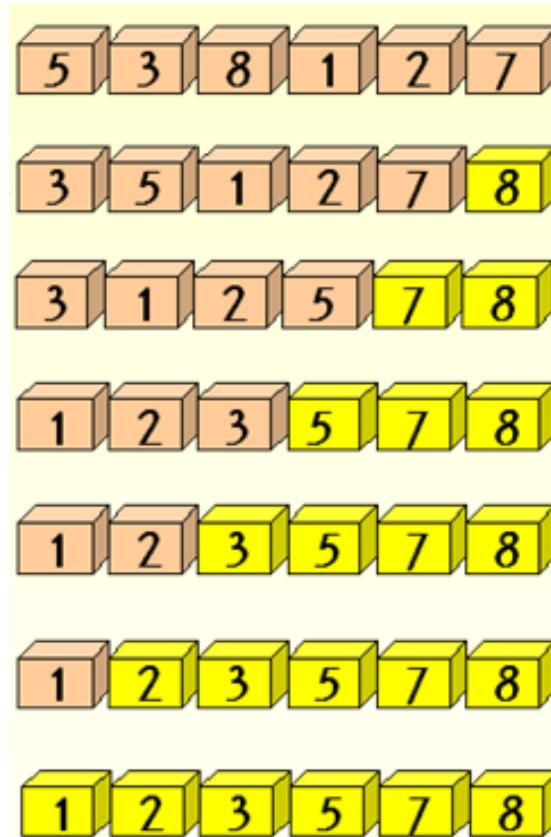


selection sort

```
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )
void selection_sort(int list[], int n)
{
    int i, j, least, temp;
    for(i=0; i<n-1; i++) {
        least = i;
        for(j=i+1; j<n; j++) // search minimum
            if(list[j]<list[least]) least = j;
        SWAP(list[i], list[least], temp);
    }
}
```

Bubble Sort

- *Compares adjacent array elements*
 - *Exchanges their values if they are *out of order**
- Smaller values *bubble up* to the top of the array
 - Larger values sink to the bottom
- Performs:
 - $O(n^2)$ comparisons



bubble_sort

```
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )
void bubble_sort(int list[], int n)
{ int i, j, temp;
  for(i=n-1; i>0; i--){
    for(j=0; j<i; j++) // compare and change
      if(list[j]>list[j+1])
        SWAP(list[j], list[j+1], temp);
  }
}
```