Algorithms and Data Structures II

Lecture 2:

Priority Queues

https://elms.u-aizu.ac.jp

Consider the situation:

- Assume that a 1000-page job and then some 1-page jobs arrive at the waiting queue of a printer .
- If the queue follows the FIFO rule then the printer will prints the 1000-page job and keeps the other 1-page jobs waiting.
- It might be better to print the 1-page jobs first and then the 1000-page job.

To realize this idea effectively,

- We need a data structure which allows the insert of a job and the deletion of the job with the minimum number of pages.
- A priority queue is a data structure for this purpose that allows at least the following two operations:
  - To insert an element into the queue;
  - To delete the maximum (or the minimum) element from the queue.

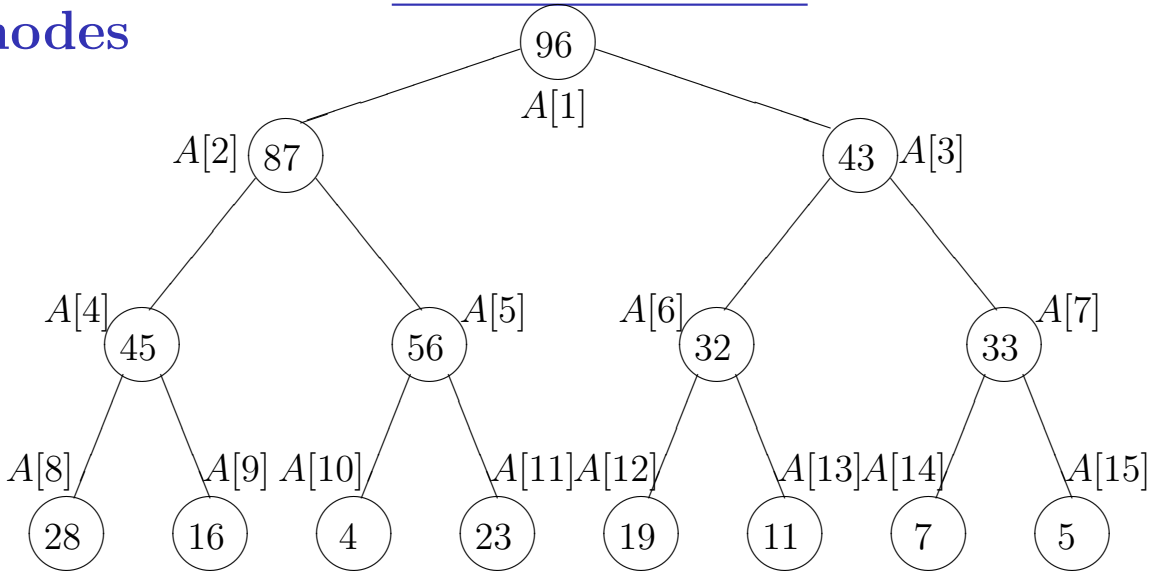
Use of priority queues

Priority queues are also used in many applications where data must be processed in order, but not necessarily in full sorted order, nor necessarily all at once. Often a set of keys must be collected, then the largest (smallest) processed, then perhaps more keys collected and the next largest (smallest) processed, and so forth.

Binary Heaps

- Binary heap (or simply heap) is an efficient implementation for priority queues.
- A maximum heap of size n is a complete binary tree of n nodes such that the key of each node is less than or equal to that of its parent (*maximum heap condition*).
- The above implies in particular that the largest key is in the root of the binary tree. (heap is not sorted!)
- The tree structure makes the maintenance cost $O(\log(n))$.

An example of maximum heap with 15 nodes



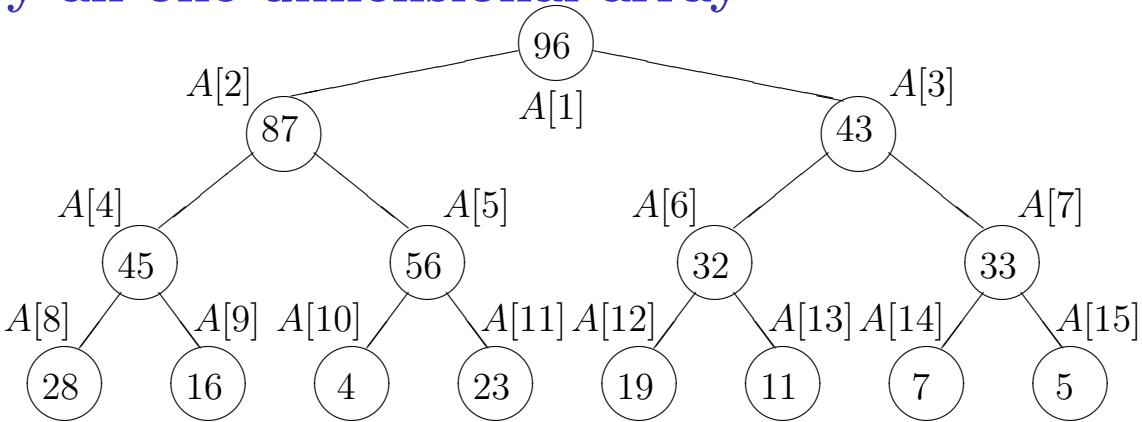
Minimum heap

- A minimum heap of size n is a complete binary tree of n nodes such that the key of each node is greater than or equal to that of its parent (minimum heap condition).
- In a minimum heap, the smallest key is in the root of the binary tree.

Representation of a complete binary tree by an one-dimensional array

- Put the root at position 1, its two children at positions 2 and 3. In general, the children of a node at position i are put at positions $2i$ (the left even-child) and $2i + 1$ (the right odd-child).
- The parent of the node at position j is at position $\lfloor j/2 \rfloor$. The root ($j = 1$) has no parent.
  - For a real number x, $\lfloor x \rfloor$ (floor or integer part of x) is the largest integer that is at most x, and $\lceil x \rceil$ (ceiling of x) is the smallest integer that is at least x.
  - Example: $\lfloor 5/2 \rfloor = \lfloor 2.5 \rfloor = 2$ whereas $\lceil 2.5 \rceil = 3$

Representation of a complete binary tree by an one-dimensional array



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
96	87	43	45	56	32	33	28	16	4	23	19	11	7	5

Some Operations on Heaps (I)

- Insert. To insert a new element into maximum heap $A[1], \ldots, A[n-1]$.
- First put the new element in $A[n]$.
- Then check if the maximum heap condition is violated. If so, the violation can be fixed by exchanging the keys at A[n] and its parent ($A[\lfloor n/2 \rfloor]$). This may, in turn, cause a violation, and thus can be fixed in the same way.
- At the end, a heap is one element large.

Some Operations on Heaps (I) cntd.

- The following gives an implementation of the above insertion. insert adds a new item to $A[n]$, then calls upheap(n) to fix the heap condition violation at position n.

upheap & insert

```
upheap(int A[], int k){
    int v;
    v = A[k];
    while (k > 1 && A[k/2] <= v){
        A[k] = A[k/2]; k = k/2;
    }
    A[k] = v;
}
insert(int A[], int v, int n){
    A[n] = v;
    upheap(A, n);
}
```

The time complexity of insert operation on a heap of size n is $O(\log n)$

Some Operations on Heaps (II)

- Replace. To replace the key at the root (the first element of A), then moving the key down from top to bottom to restore the heap condition.

  Process of repeatedly exchanging of a key with the largest of his children until the heap condition is restored or the bottom of the tree is reached.

downheap & replace

```
downheap(int A[ ], int k, int n){
  int j, v;
  v = A[k];
  while (k <= n/2) {
    j = k+k;
    if (j<n && A[j]<A[j+1]) j++;
    if (v >= A[j]) break;
    A[k]=A[j]; k=j; }
  A[k] = v;
}
replace(int v){
  A[1]=v;
  downheap(A, 1, n);}
```

The time complexity of insert replace on a heap of size n is $O(\log n)$

Some Operations on Heaps (III)

Delete the maximum key. To delete the largest key from the root of the heap and re- arranging the remaining keys into a heap.

```
static int n;
int delete_max(int A[]){
  int v = A[1];
  A[1] = A[n]; n--;
  downheap(A, 1, n);
  return v;
}
```

The time complexity of insert delete on a heap of size n is $O(\log n)$

Some Operations on Heaps (IV)

Construct a heap. A heap of size n can be constructed by inserting n keys into an array (from $A[1]$ to $A[n]$). In the case that the keys are already in $A[1], \ldots, A[n]$, we can re- arrange the keys in A into a heap by calling upheap.

```
construct_1(int A[], int n){
  int i;
  for (i = 1; i <= n; i++)
    upheap(A,i);
}
```

The above method takes $O(n \log n)$ time to build a heap.

construct_2

We now introduce a more efficient way which takes $O(n)$ time to construct a heap.

```
construct_2(int A[], int n){ int i;
  for (i = n/2; i >= 1; i--)
    downheap(A, i, n);
}
```

The cost : $n/2$ times loop of downheap with $O(\log n)$ cost.

Why $O(n)$? on construct_2, not $O(n \log n)$?

- Assume $N = n + 1 = 2^h$ for simplicity, $h = \log_2 N$.
- The number of replacement will be

$$
\begin{aligned}
& 1 \cdot \frac{N}{2^2} + 2 \cdot \frac{N}{2^3} + 3 \cdot \frac{N}{2^4} + \cdots + (h-1) \cdot \frac{N}{2^h} \\
= & \sum_{k=1}^{h-1} \frac{kN}{2^{k+1}} = \frac{N}{4} \sum_{k=1}^{h-1} \frac{k}{2^{k-1}} \leq \frac{N}{4} \sum_{k=1}^{\infty} k x^{k-1} \Big|_{x=\frac{1}{2}} \\
= & \frac{N}{4} \frac{d}{dx} \left(\sum_{k=1}^{\infty} x^k \right) \Big|_{x=\frac{1}{2}} = \frac{N}{4} \frac{d}{dx} \left(\frac{1}{1-x} \right) \Big|_{x=\frac{1}{2}} \\
= & \frac{N}{4} \frac{1}{(1-x)^2} \Big|_{x=\frac{1}{2}} = N
\end{aligned}
$$

Heap Sort

- The heap sort can be realized by first making the n elements into a heap and then deleting the largest element, making the remaining elements into a heap, then deleting the next largest, until the elements are sorted.

Idea for heap sort

1. construct_2
2.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
96	87	43	45	56	32	33	28	16	4	23	19	11	7	5

3. tmp = A[1]; A[1] = A[n]; A[n] = tmp;
4.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
5	87	43	45	56	32	33	28	16	4	23	19	11	7	96

5. downheap
6.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
87	56	43	45	23	32	33	28	16	4	5	19	11	7	96

7. ⋮

Heap Sort

```
heapsort(int A[], int n){
  int i, tmp;
  construct_2(A, n);
  for (i = n; i >=2; i--){
    tmp=A[1]; A[1] = A[i]; A[i] = tmp;
    downheap(A, 1, i-1);
}
```

The time complexity of heap sort of size n is $O(n \log n)$.