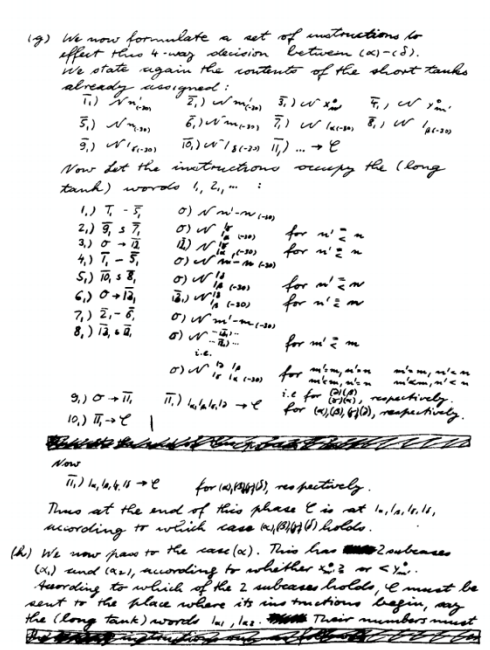
CS3334 Data Structures

Lecture 7: Heap, Bucket & Radix Sorts



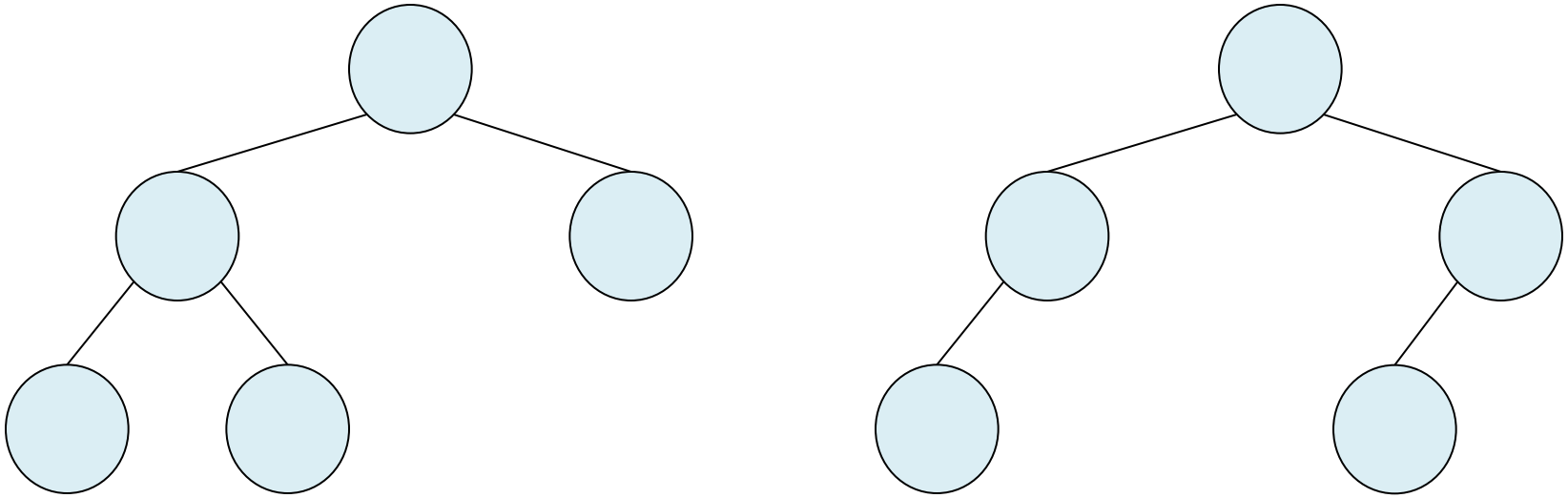
Chee Wei Tan

Story Thus Far

- Breaking the Ω(nlogn) barrier in comparison-based sorting algorithms for sorting
- Know abstract data structures fundamental to many mathematical objects, e.g., permutation, natural numbers
- Know algorithms for order statistics
  - Instrumental for selecting ideal pivots in partitioning arrays, e.g., selecting the median, the smallest or largest item, the *k*th smallest item, partial sorting
  - *Selection* is fundamental to Divide-and-conquer ideas
    - *Median of the medians, Blum, Floyd, Pratt, Rivest, Tarjan (1973)*

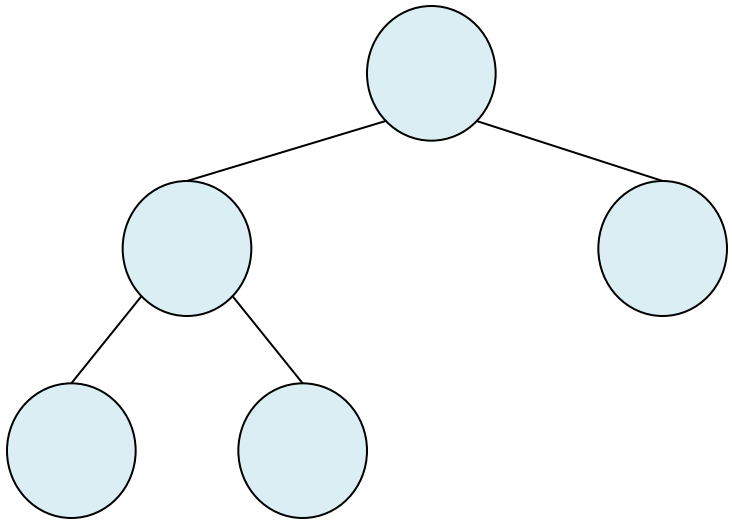
Binary Tree

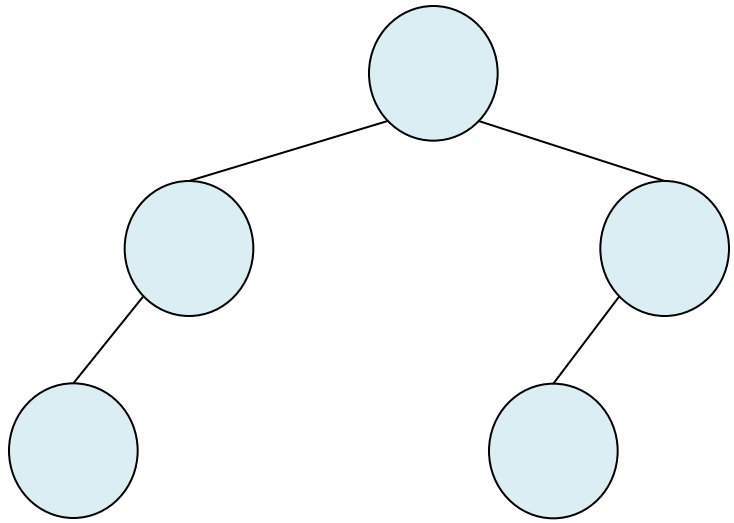
- It is a tree in which every node has at most two children.
- For example,



Complete Binary Tree (1/4)

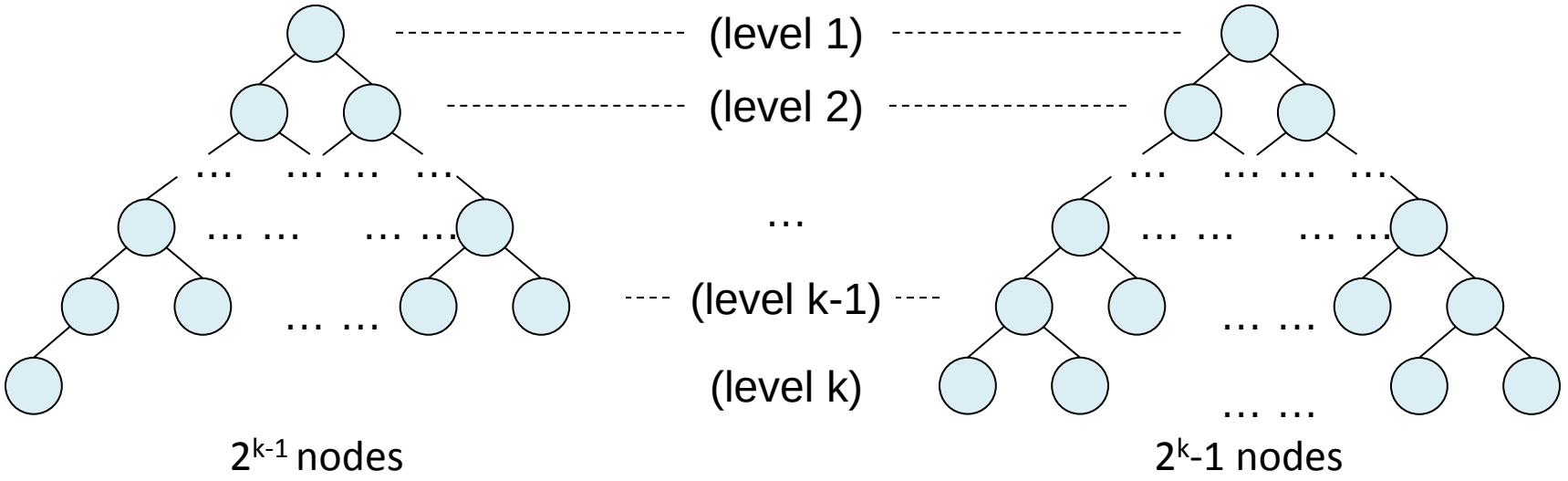
- Complete binary tree: A binary tree with all levels completely filled except possibly the bottom level, which is filled from left to right.
- Q: Which one is a complete binary tree?





Complete Binary Tree (2/4)

- A complete binary tree with k levels
  - The minimum number of nodes is 2^{k-1}
  - The maximum number of nodes is 2^k-1
- Let n = number of nodes. Then, $2^{k-1} \le n \le 2^k-1$.

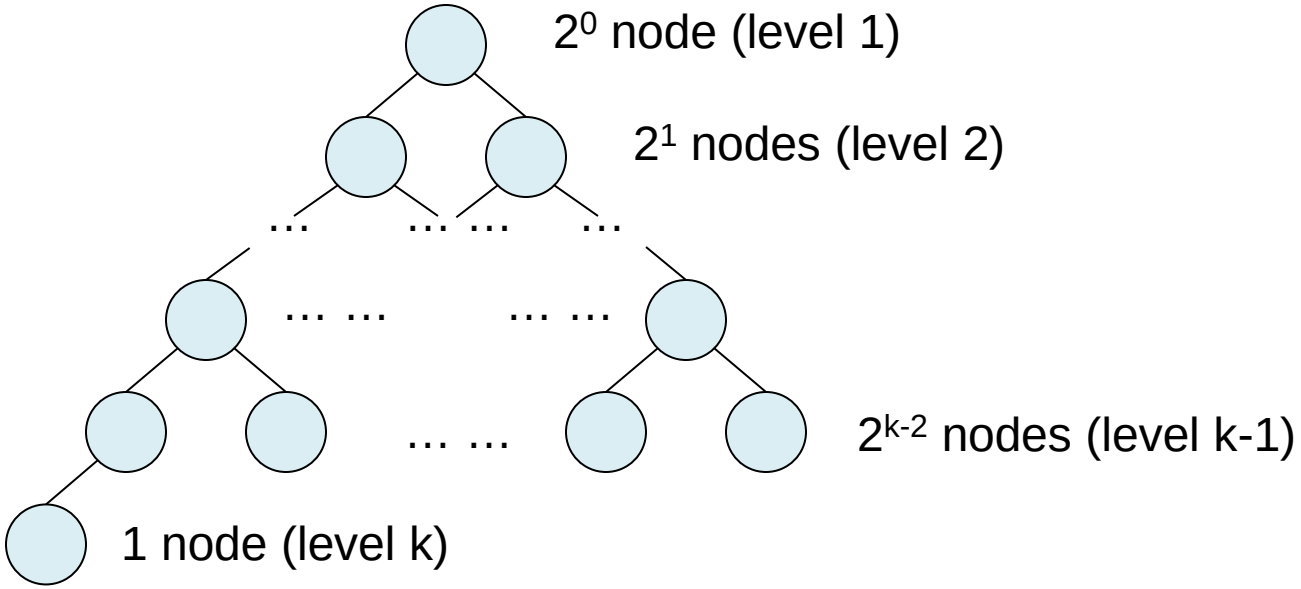


Complete Binary Tree (3/4)

- The **minimum** number of nodes of a complete binary tree with k levels is 2^{k-1}

$= 2^0 + 2^1 + \ldots + 2^{k-2} + 1 = (2^{k-1} - 1)/(2 - 1) + 1 = 2^{k-1}$

(Note: $ar^0+ar^1+ar^2+ \ldots +ar^n = a(r^{n+1}-1)(r-1)$, if $r>1$)

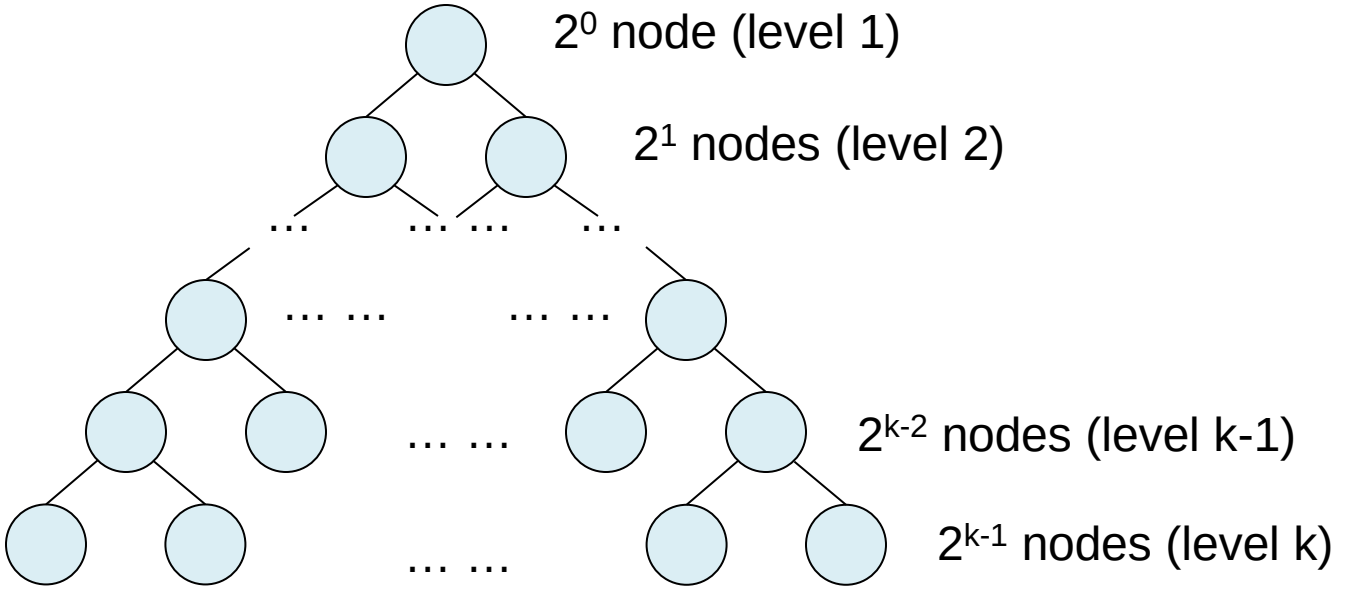


Complete Binary Tree (4/4)

- The **maximum** number of nodes of a complete binary tree with k levels is 2^k-1

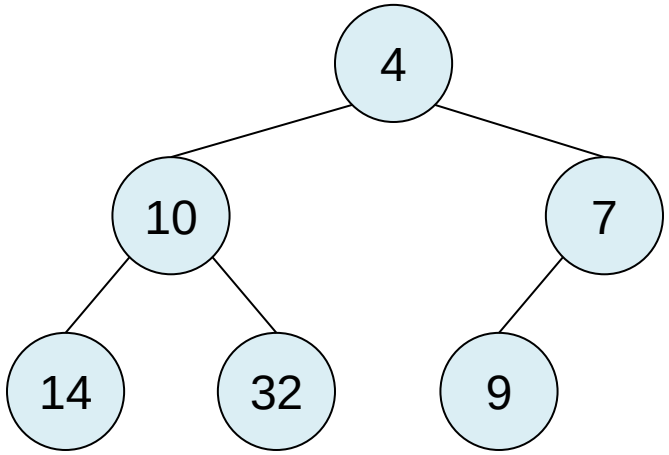
$= 2^0+ 2^1 + \ldots + 2^{k-2} + 2^{k-1} = (2^k - 1)/(2 - 1) = 2^k-1$

(Note: $ar^0+ar^1+ar^2+ \ldots +ar^n = a(r^{n+1}-1)(r-1)$, if $r>1$)



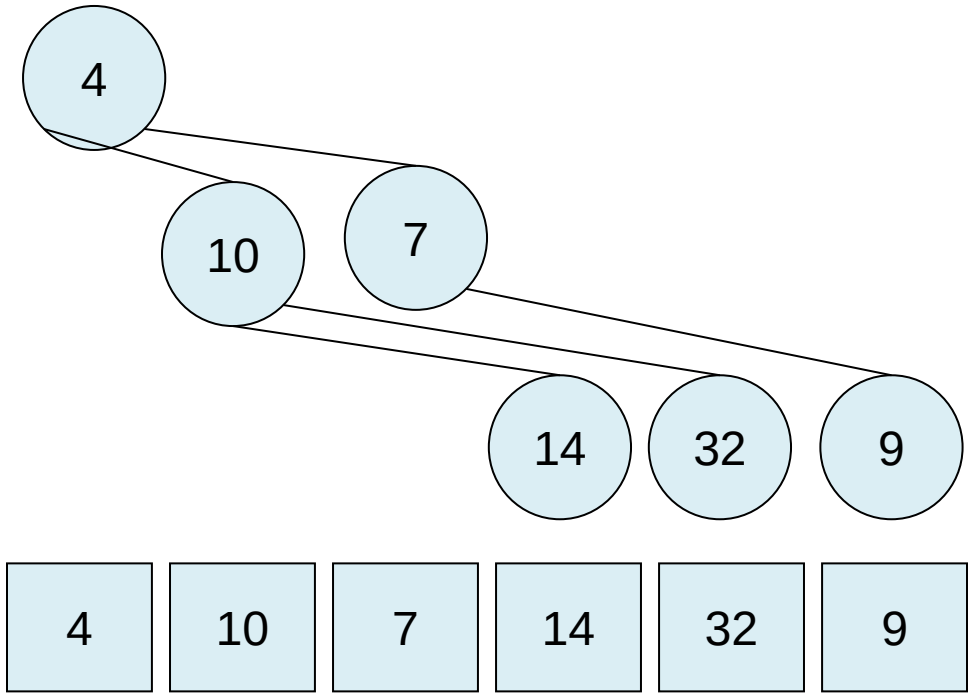
Min-Heap

- Elements “conceptually” stored in a complete binary tree, one element per node, so that the **min-heap property** is satisfied if
  - Value in any node (except the root) is ≥ value in its parent, and
  - The smallest value is at the root.



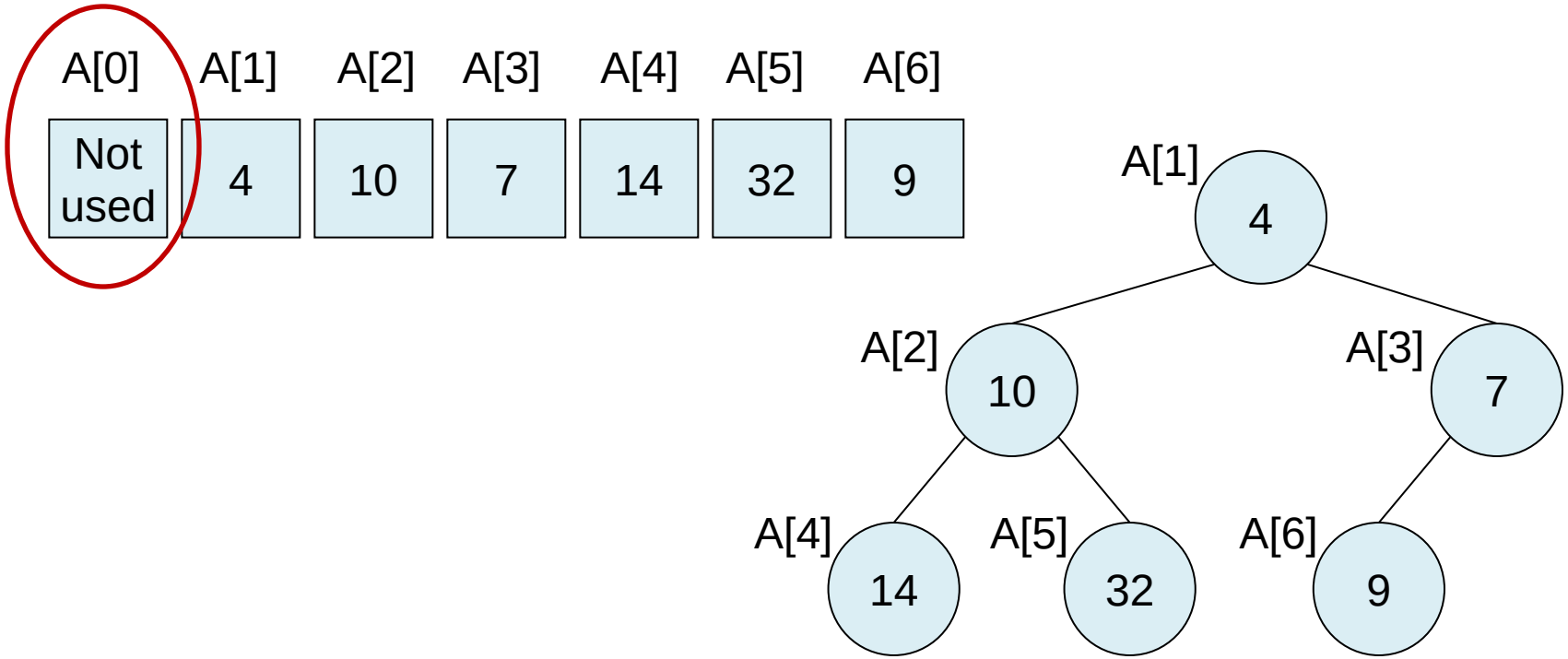
Implementation of Binary Heap

- Binary tree is an abstract data structure (mental construct for mathematical study)
- Encode the heap as an array (conducive for fast implementation in physical machines)



Implementation of Binary Heap

- Root stored in A[1], NOT A[0]
- Children of A[i] are stored in A[2i] and A[2i+1]
- Parent of A[i] is stored in A[⌊i/2⌋]

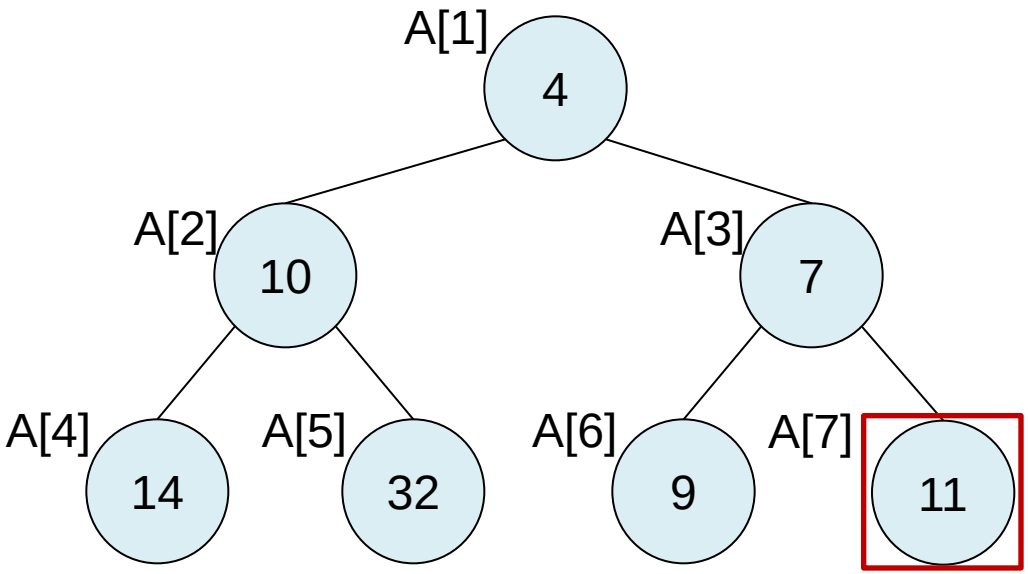


Insertion

- insert(x):
  - Create a node at the end (the leftmost hole in the bottom level)
  - If x < parent(x), then swap x and parent(x)
  - Set x = parent(x) and repeat the previous step until no violation of the min-heap property
- After swapping x and parent(x), we need not check parent(x) and sibling(x). Why?

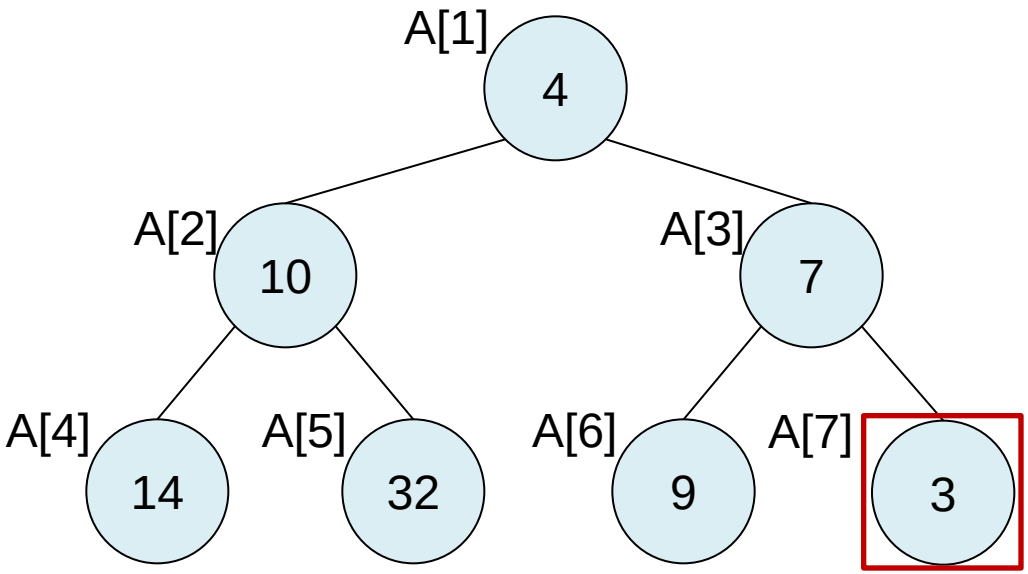
Insertion (Example 1)

- Insert 11, compare 11 & 7, and no swapping is needed



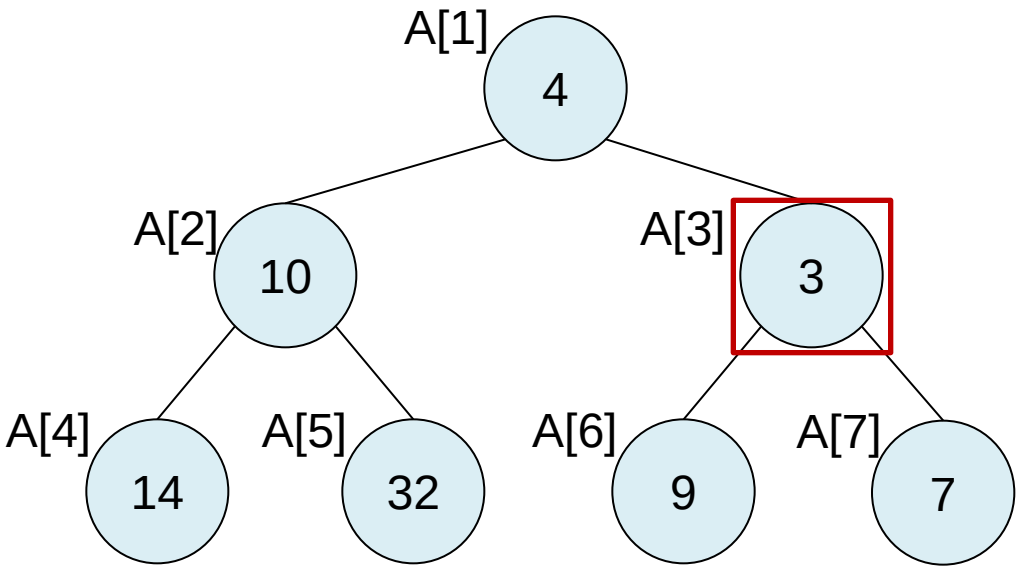
Insertion (Example 2) (1/3)

- Insert 3, compare 3 & 7, and swap 3 & 7



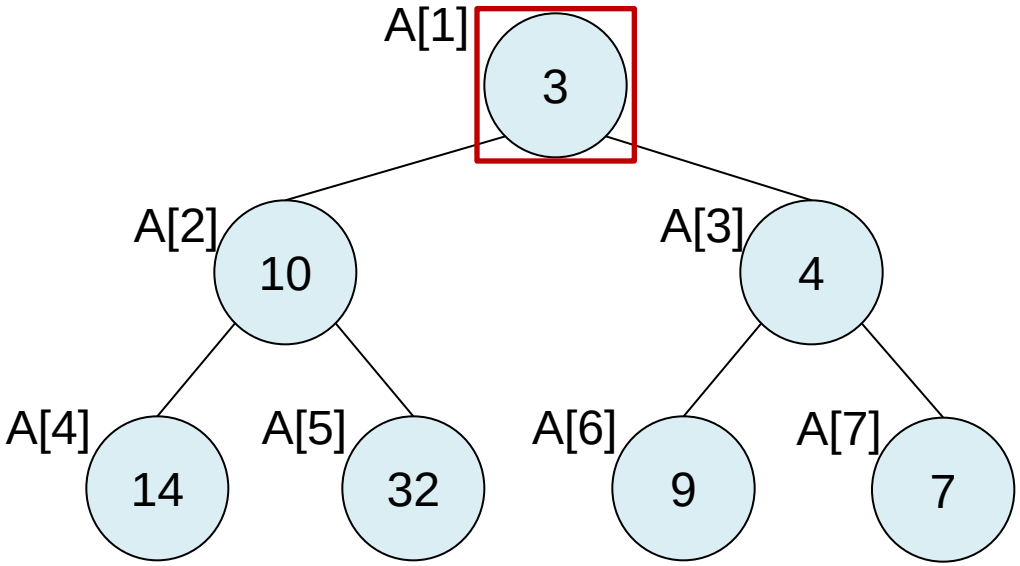
Insertion (Example 2) (2/3)

- Compare 3 & 4 and swap 3 & 4



Insertion (Example 2) (3/3)

- Done



Code

- Assume the following global variables:

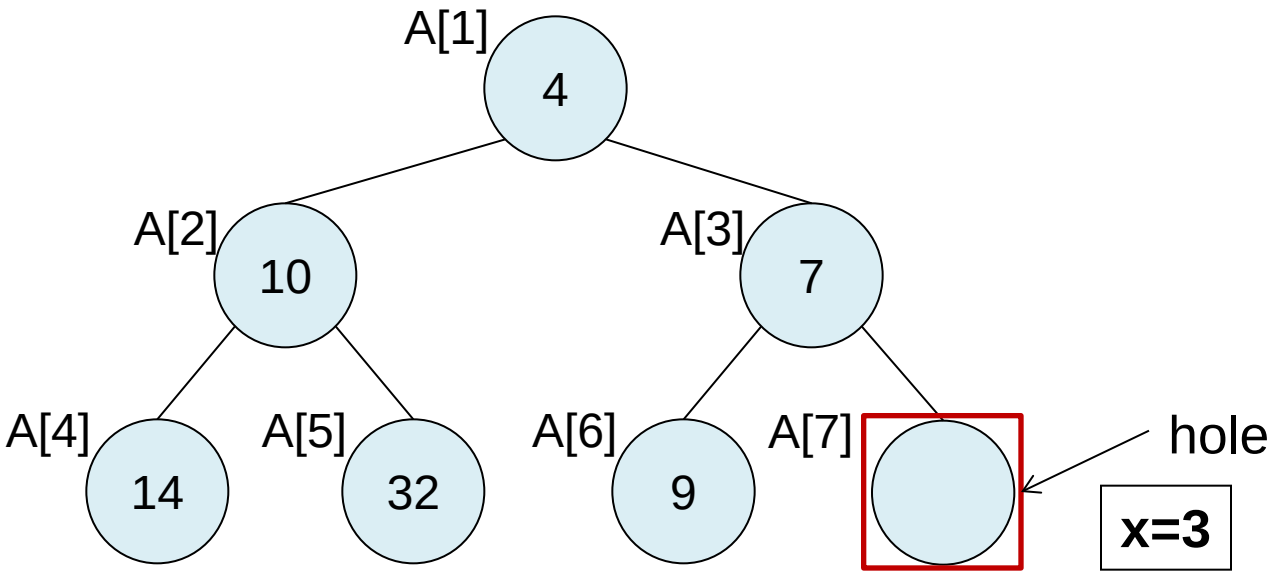
  `item A[0..MAXSIZE]` and `int size`

```
void insert(item x)
{
   size++;
   int hole = size;
   A[hole] = x; // next available slot in A[]

   while (hole>1 && A[hole]<A[hole/2])
   {
     item temp = A[hole]; // swap A[hole] & A[hole/2]
     A[hole] = A[hole/2];
     A[hole/2] = temp;
     hole = hole/2; // set hole to parent(hole)
   }
}
```

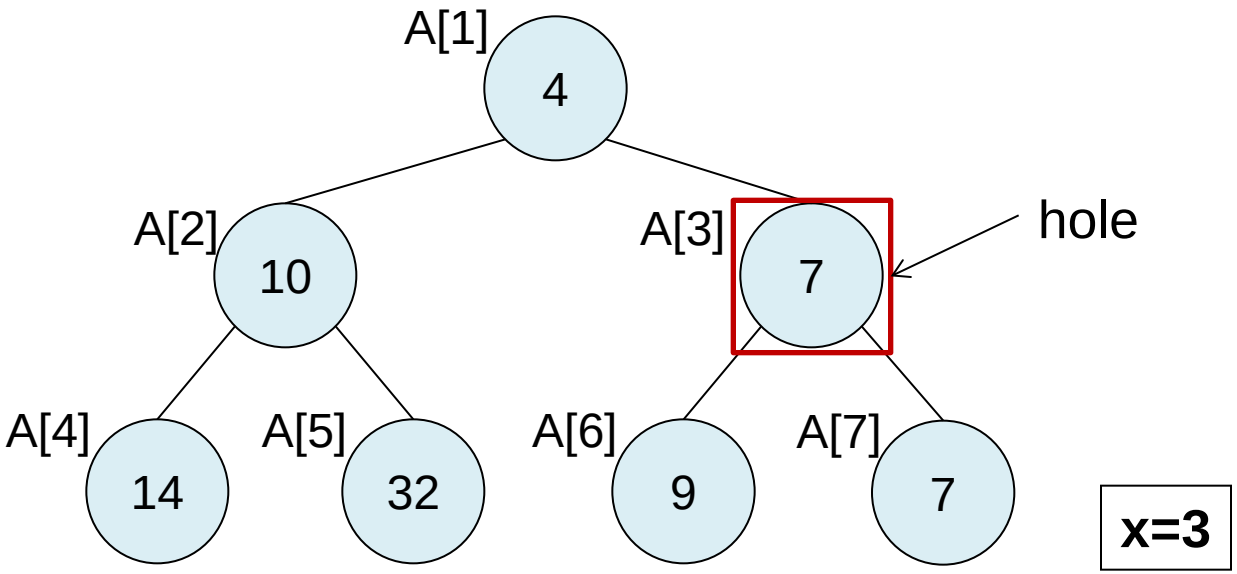
Smarter Insertion (Example 3) (1/3)

- Compare x=3 & 7, move 7 to A[hole], and set hole to $\lfloor 7/2 \rfloor$



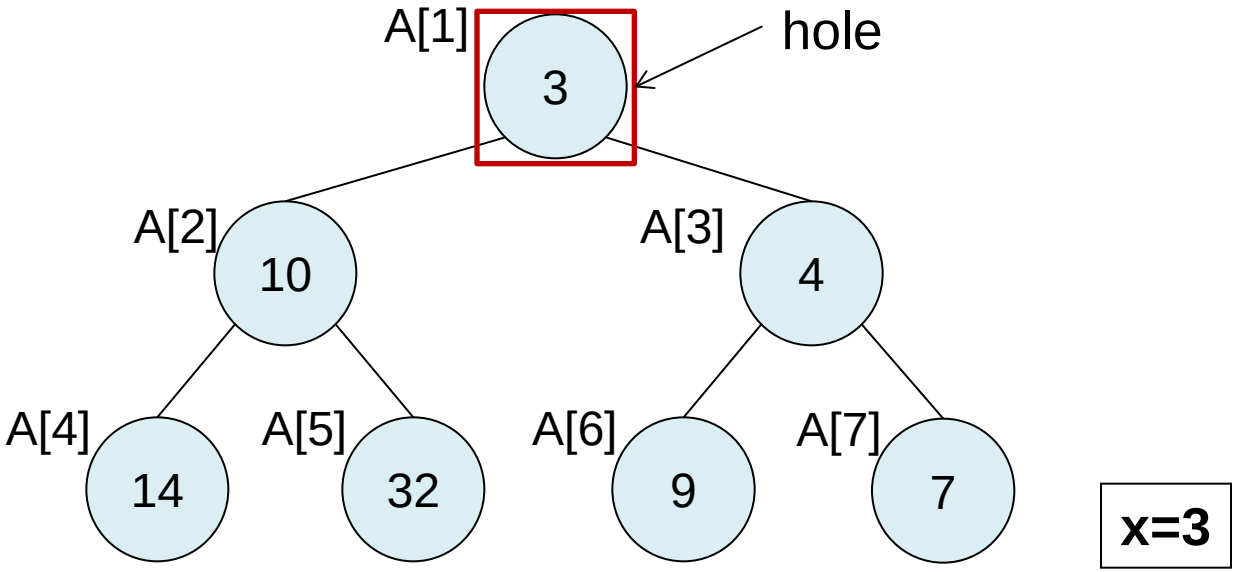
Smarter Insertion (Example 3) (2/3)

- Compare x=3 & 4, move 4 to A[hole], and set hole to $\lfloor 3/2 \rfloor$



Smarter Insertion (Example 3) (3/3)

- Since hole = 1, copy x=3 to A[1]



Code (Smarter Version)

- Find the right place for x before storing x

```
void insert(item x)
{
   size++;
   int hole = size;
   while (hole>1 && x<A[hole/2])
   {
      A[hole]=A[hole/2];
      hole = hole/2;
   }
   A[hole]=x;
}
```

Insertion

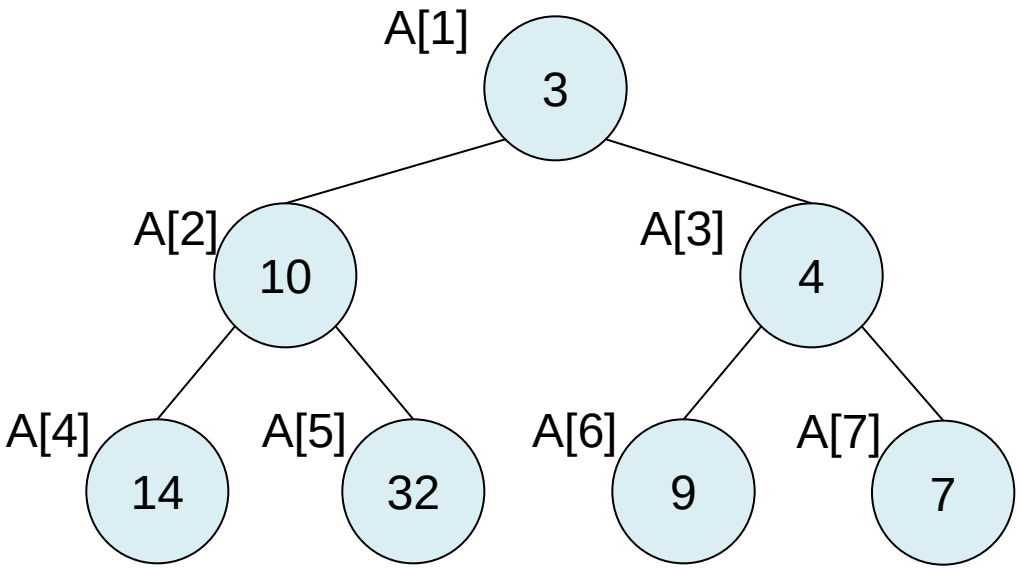
- Worst-case time complexity:
  - If the new element is the new minimum, it will be pushed up to root.
  - For a complete binary tree with n nodes and k levels, i.e., $2^{k-1} \le n \le 2^k - 1$.
  - So, $2^{k-1} \le n \Rightarrow$
    $\log_2 2^{k-1} \le \log_2 n \Rightarrow$
    $k-1 \le \log_2 n \Rightarrow$
    $k \le \log_2 n + 1$
  - Therefore, insertion takes O(logn) time in the worst case.

Deletion

- deleteMin(x):
  - Remove the root (minimum) and store it into x
  - Place the last element, y, at root
  - Push y down to its correct position
- Worst-case time complexity
  - If y is the largest element, it will be pushed down to the bottom level
  - O(logn) in the worst case

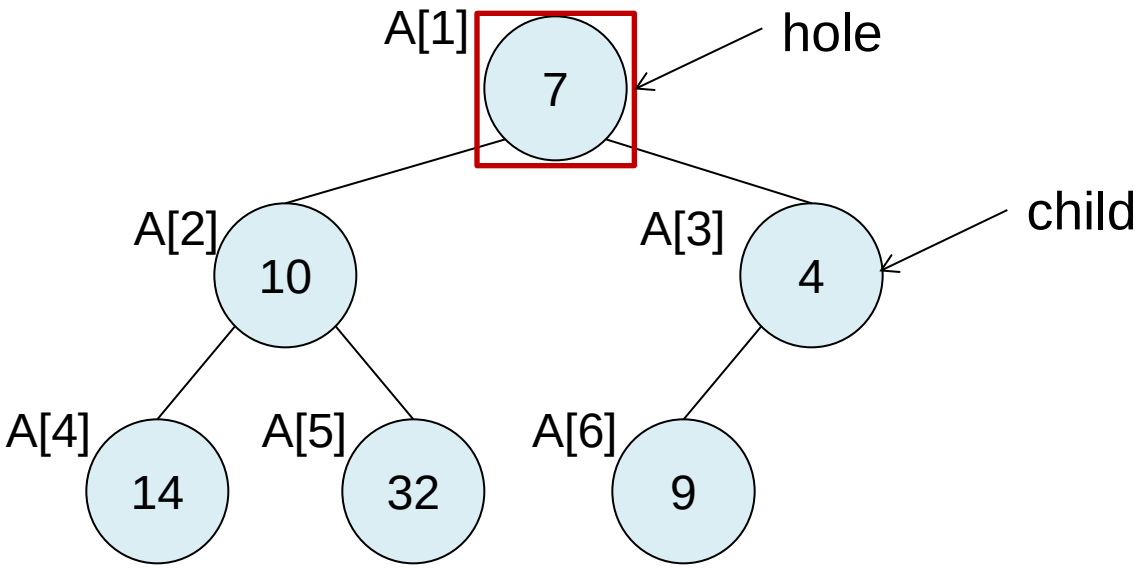
Deletion (Example 4) (1/3)

- Set x=3, remove 3, and place the last element 7 at root



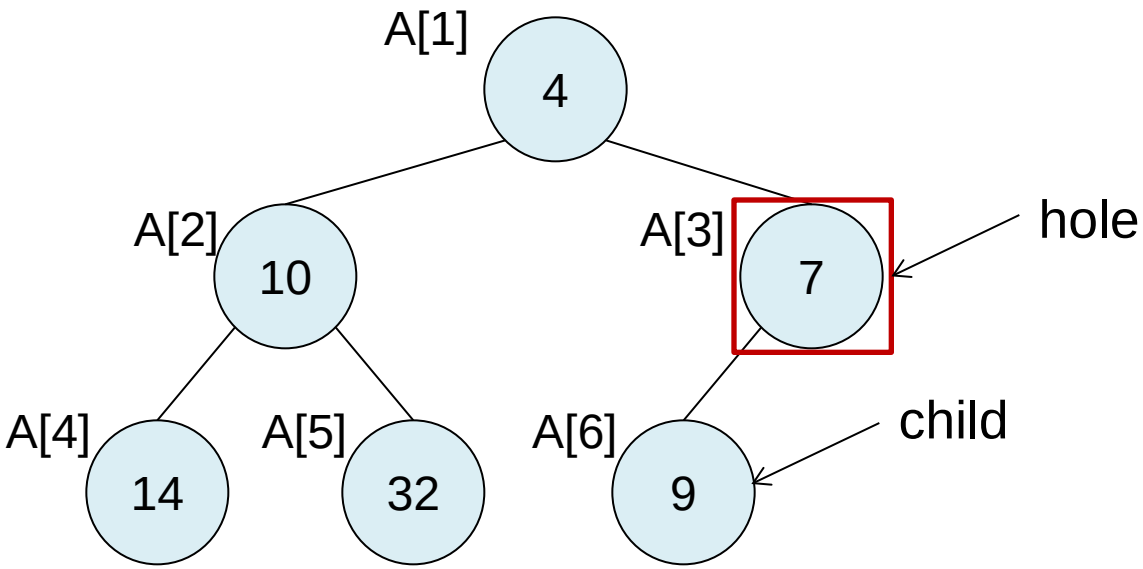
Deletion (Example 4) (2/3)

- Compare 7 & 4 and swap 7 & 4



Deletion (Example 4) (3/3)

- Compare 7 & 9, and no further swapping



Code (1/2)

```
void deleteMin(item &x)
{
   x = A[1];
   A[1] = A[size];
   size--;
   pushDown(1);
}
```

Code (2/2)

```
// to push A[i] down to its correct position
void pushDown(int i)
{
   item y=A[i];
   int hole=i, child=2*hole;
   while (child<=size)
   {
      if (child<size && A[child]>A[child+1])
         child++;   // child points to smaller one
      if (A[child]>=y) break;
      A[hole] = A[child];
      hole = child;
      child = 2*hole; // point to the left child
   }

   A[hole]=y;
}
```

Build a Min-Heap

- To construct a min-heap of n elements (i.e., buildHeap())
- For each non-leaf node i from bottom to root, call pushDown(i)
  - A complete binary tree with n nodes has $\lfloor n/2 \rfloor$ leaves
  - The last node in the array which has a child is at position $\lfloor n/2 \rfloor$
  - The time complexity is O(n) in the worst case

```
// to arrange A[1..n] in heap order
void buildHeap(int n)
{
   size=n;
   for (i=n/2; i>=1; i--)
      pushDown(i);
}
```

Heap Sort

- Heap Sort
  - Build a min-heap on the n elements, i.e., O(n) time
  - Repeat to remove the minimum element n times, i.e., O(n·logn) time
- Complexity: O(n + n·logn) = O(nlogn)

Code

```
void heapSort(int n)
// To sort A[1..n] in non-increasing order
{
   buildHeap(n);
   for (int i=n; i>=2; i--)
   {
      item temp;
      deleteMin(temp);
      A[i]=temp;
   }
}
```

More Heaps

- Max-heap
  - Analogous to min-heap where each node has two (or fewer) children, and the key of each node (i.e. the number inside the node) is greater than the keys of its child nodes.
- How to build Max-heap from Min-heap?
- Can you think of a mathematical object whose structure can be viewed as a heap?
  - If yes, what questions might be contemplated?

Bucket Sort (1/2)

- Comparison-based sorting algorithms require Ω(nlogn) time
- But we can sort in O(n) time using more powerful operations
  - When elements are integers in {0,…, M-1}, bucket sort needs O(M+n) time and O(M) space
  - When M=O(n), bucket sort needs O(2n)=O(n) time
- Note: Some books call this *counting sort*

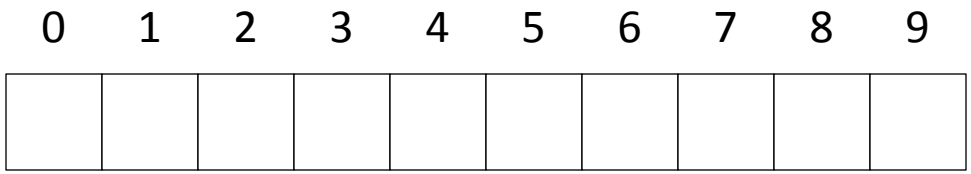
Bucket Sort (2/2)

- Idea: Require a counter (auxiliary) array C[0..M-1] to count the number of occurrences of each integer in {0,…,M-1}
- Algorithm:
  - **Step 1:** initialize all entries in C[0..M-1] to 0
  - **Step 2:** For i=0 to n-1
    - Use A[i] as an array index and increase C[A[i]] by one
  - **Step 3:** For j=0 to M-1
    - Write C[j] copies of value j into appropriate places in A[0..n-1]

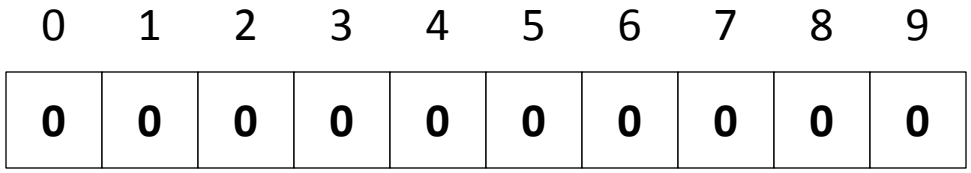
Bucket Sort (Example)

- Input: 3, 4, 6, 9, 4, 3 where M=10

Counter array A:

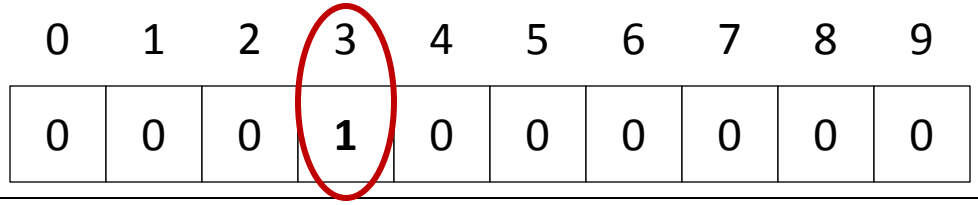


- Step 1: Initialization



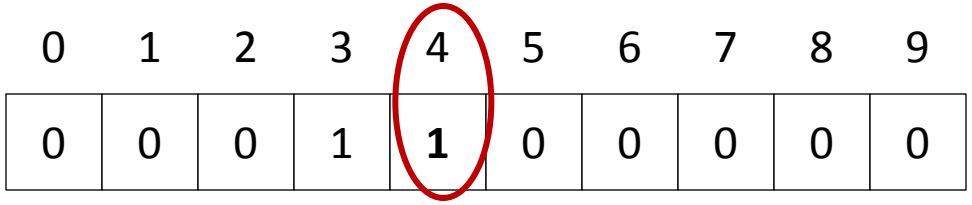
- Step 2: Read 3

(A[3] = A[3] + 1)



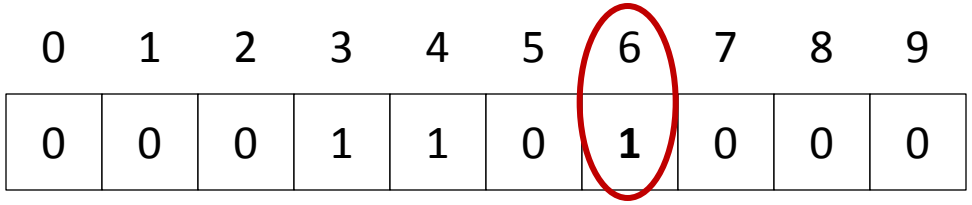
- Read 4

(A[4] = A[4] + 1)



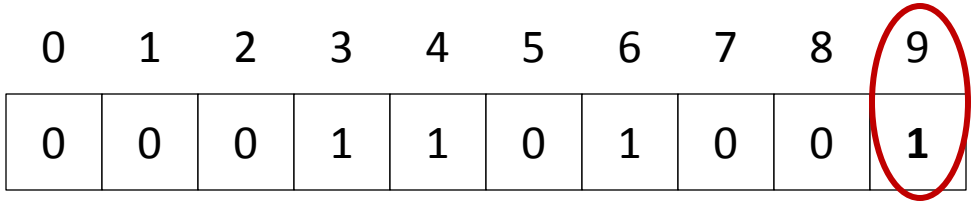
- Read 6

(A[6] = A[6] + 1)



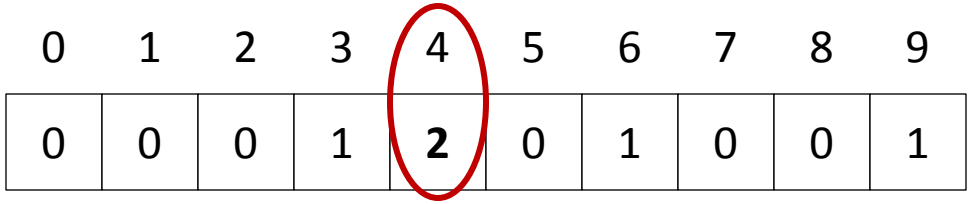
- Read 9

(A[9] = A[9] + 1)



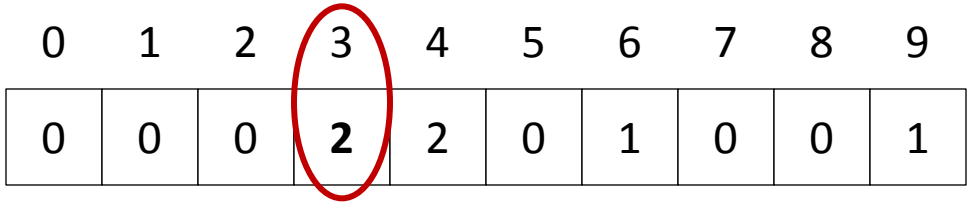
- Read 4

(A[4] = A[4] + 1)



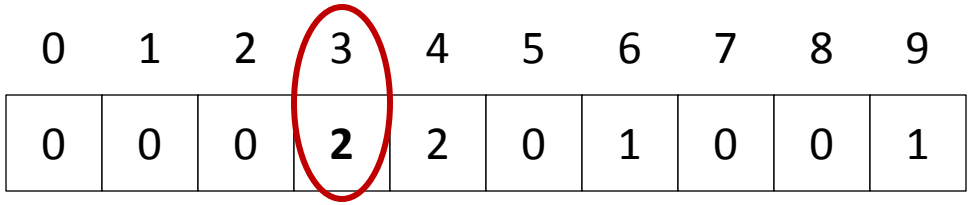
- Read 3

(A[3] = A[3] + 1)

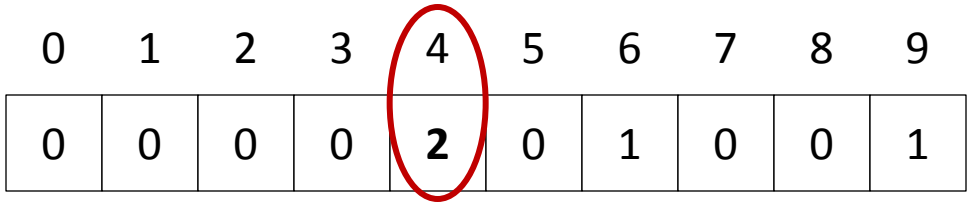


- Step 3: Print the result (from index 0 to 9)

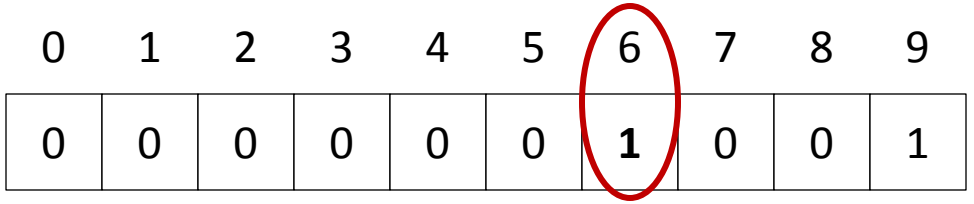
- Result: 3, 3



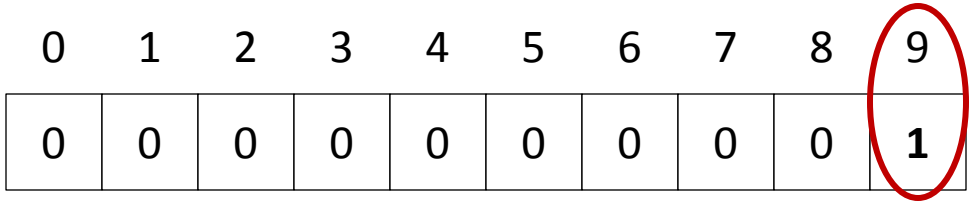
- Result: 3, 3, 4, 4



- Result: 3, 3, 4, 4, 6



- Result: 3, 3, 4, 4, 6, 9



Radix Sort

- Bucket sort is not efficient if M is large
- The idea of radix sort:
  - Apply bucket sort on each digit (from Least Significant Digit to Most Significant Digit)
- A complication:
  - Just keeping the count is not enough
  - Need to keep the actual elements
  - Use a queue for each digit

Radix Sort (Example) (1/3)

- Input: 17<u>0</u>, 04<u>5</u>, 07<u>5</u>, 09<u>0</u>, 00<u>2</u>, 02<u>4</u>, 80<u>2</u>, 06<u>6</u>
- The **first** pass
  - Consider **the least significant digits** as keys and move the keys into their buckets

0	17<u>0</u>, 09<u>0</u>
1	
2	00<u>2</u>, 80<u>2</u>
3	
4	02<u>4</u>
5	04<u>5</u>, 07<u>5</u>
6	06<u>6</u>
7	
8	
9	

  - Output: 17<u>0</u>, 09<u>0</u>, 00<u>2</u>, 80<u>2</u>, 02<u>4</u>, 04<u>5</u>, 07<u>5</u>, 06<u>6</u>

Radix Sort (Example) (2/3)

- The **second** pass
- Input: 170, 090, 002, 802, 024, 045, 075, 066
  - Consider **the second least significant digits** as keys and move the keys into their buckets

0	002, 802
1	
2	024
3	
4	045
5	
6	066
7	170, 075
8	
9	090

  - Output: 002, 802, 024, 045, 066, 170, 075, 090

Radix Sort (Example) (3/3)

- The **third** pass
- Input: 002, 802, 024, 045, 066, 170, 075, 090
  - Consider **the third least significant digits** as keys and move the keys into their buckets

0	002, 024, 045, 066, 075, 090
1	170
2	
3	
4	
5	
6	
7	
8	802
9	

  - Output: 002, 024, 045, 066, 075, 090, 170, 802 (Sorted)

Code (1/2)

```
// item is the type: {0,…,10^d-1},
// i.e., the type of d-digit integers
void radixsort(item A[], int n, int d)
{
   int i;
   for (i=0; i<d; i++)
      bucketsort(A, n, i);
}

// To extract d-th digit of x
int digit(item x, int d)
{
   int i;
   for (i=0; i<d; i++)
      x /= 10; // integer division
   return x%10;
}
```

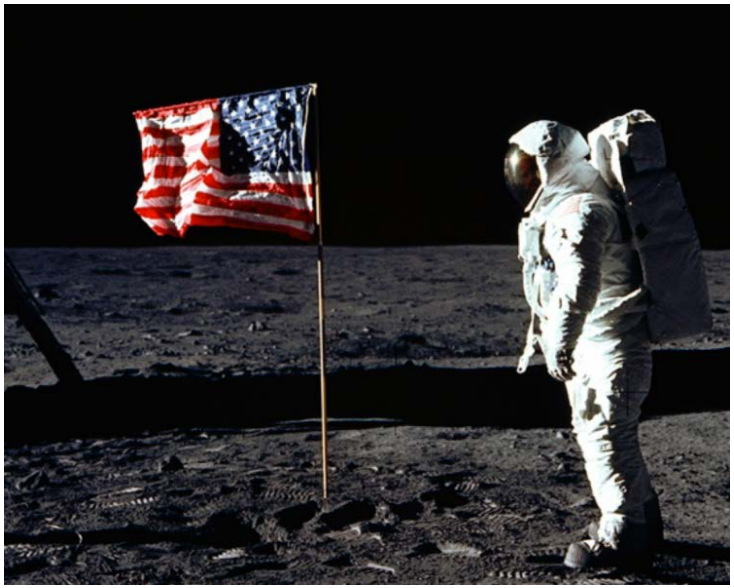
Code (2/2)

```
void bucketsort(item A[], int n, int d)
// stable-sort according to d-th digit
{
   int i, j;
   Queue *C = new Queue[10];
   for (i=0; i<10; i++) C[i].makeEmpty();
   for (i=0; i<n; i++)
      C[digit(A[i],d)].EnQueue(A[i]);
   for (i=0, j=0; i<10; i++)
      while (!C[i].empty())
      { // copy values from queues to A[]
         C[i].DeQueue(A[j]);
         j++;
      }
}
```

Worst-case Time Complexity

- Assume d digits, each digit comes from {0,…,M-1}
- For each digit,
  - O(M) time to initialize M queues,
  - O(n) time to distribute n numbers into M queues
- Total time = O(d(M+n))
- When d is constant and M = O(n), we can make radix sort run in linear time, i.e., O(n).

The American Flag Sorting Problem



The name comes by analogy with the Dutch national flag problem (see previous lecture): efficiently partition the array into many "stripes".

McIlroy, Peter M.; Bostic, Keith; McIlroy, M. Douglas (1993). "Engineering radix sort". *Computing Systems*. **6** (1): 5–27.

Engineering Radix Sort

Peter M. McIlroy

Keith Bostic

Computer Science Research Group
University of California at Berkeley

M. Douglas McIlroy

AT&T Bell Laboratories

ABSTRACT

Radix sorting methods have excellent asymptotic performance on string data, for which comparison is not a unit-time operation. Attractive for use in large byte-addressable memories, these methods have nevertheless long been eclipsed by more easily programmed algorithms. Three ways to sort strings by bytes left to right—a stable list sort, a stable two-array sort, and an in-place ‘‘American flag’’ sort—are illustrated with practical C programs. For heavy-duty sorting, all three perform comparably, usually running at least twice as fast as a good quicksort. We recommend American flag sort for general use.

1. Introduction

For sorting strings you can’t beat radix sort—or so the theory says. The idea is simple. Deal the strings into piles by their first letters. One pile gets all the empty strings. The next gets all the strings that begin with *A*-; another gets *B*- strings, and so on. Split these piles recursively on second and further letters until the strings end. When there are no more piles to split, pick up all the piles in order. The strings are sorted.

In theory radix sort is perfectly efficient. It looks at just enough letters in each string to distinguish it from all the rest. There is no way to inspect fewer letters and still be sure that the strings are properly sorted. But this theory doesn’t tell the whole story: it’s hard to keep track of the piles.

Our main concern is bookkeeping, which can make or break radix sorting as a practical method. The paper may be read as a thorough answer to excercises posed in Knuth chapters 5.2 and 5.2.5, where the general plan is laid out.[1] Knuth also describes the other classical sorting methods that we refer to: radix exchange, quicksort, insertion sort, Shell sort, and little-endian radix sort.