CS32 Week 7: Hash table & Heap

Hash table

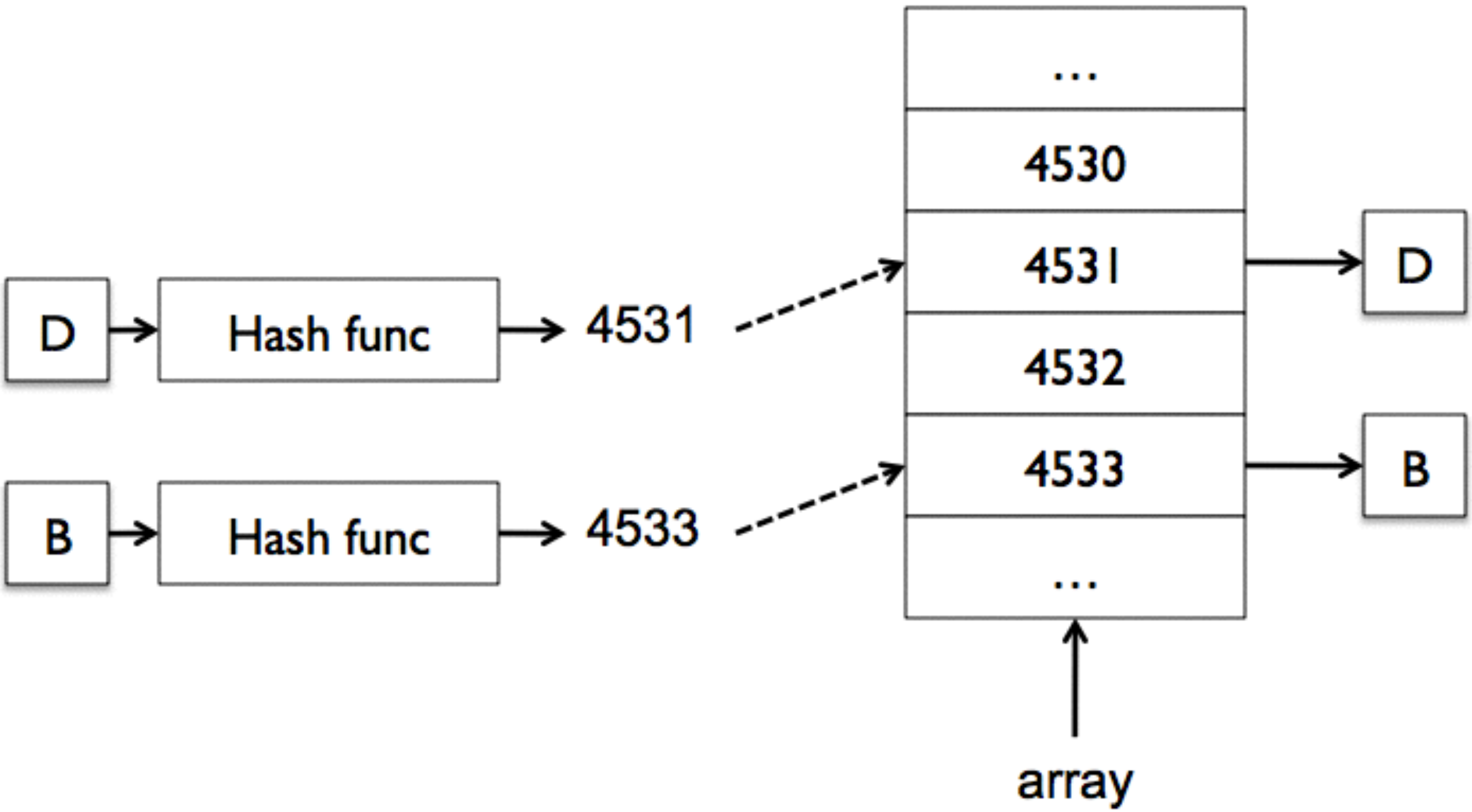
- Hash function
  - "Carey" -> H(x) -> 4531

  - Basic requirement:
    - For same key, produce same value.
  - Better hash function:
    - Spreads out the values: two different keys are likely to result in different hash values.
    - Computes each value quickly.

Hash table

- If we have a perfect hash function:
  - H(x) could map the key into an integer range of [0, 10000]
  - different key will result in different hash value.
- We could use the hash function to store the data to support fast retrieval.

Hash table



Hash table

- Time complexity:
  - Insert
    - O(1)
  - Delete
    - O(1)
  - Search
    - Compute the hash value for the key
    - Go to the memory location
    - O(1)

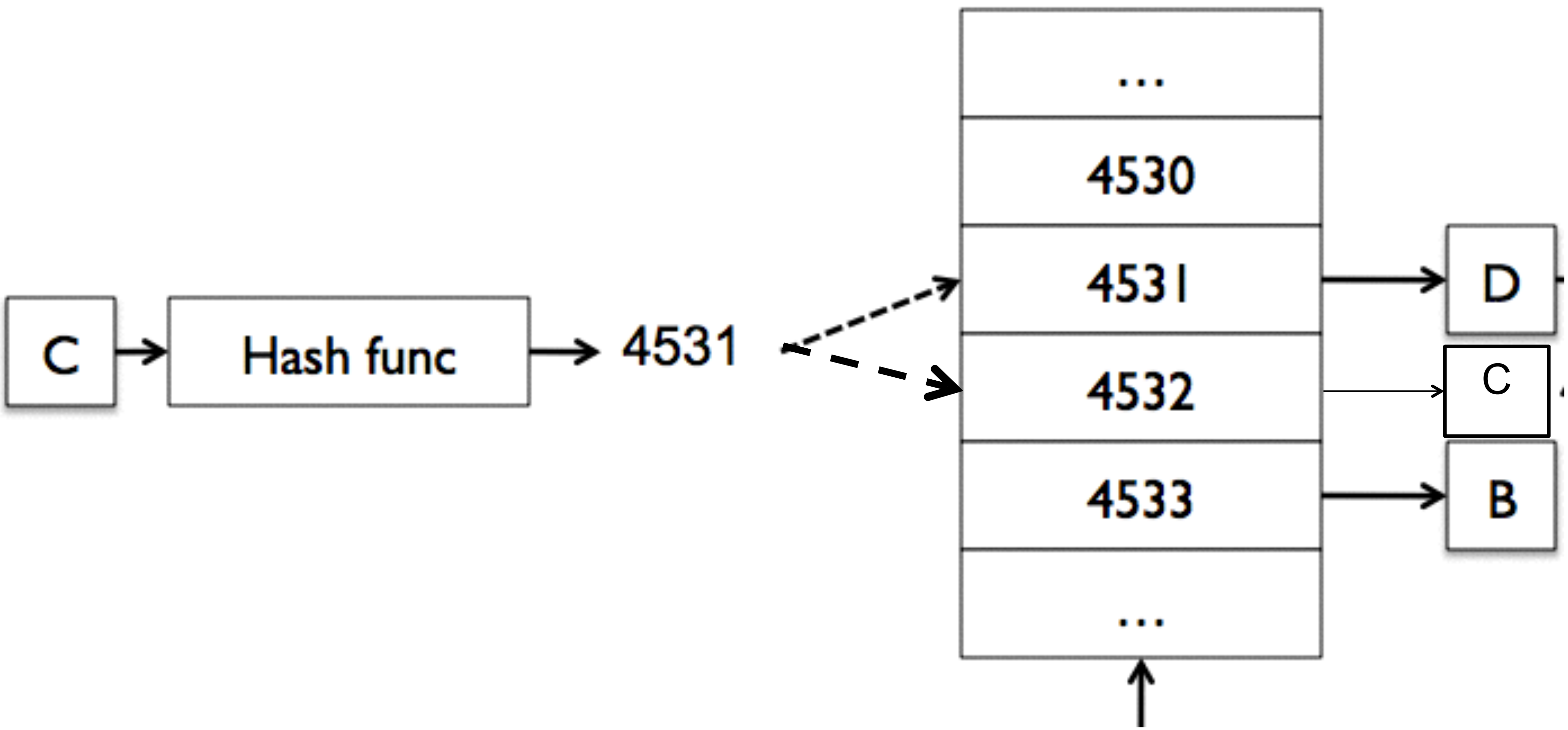
Hash table

- But there is no perfect hash functions
  - There exists the case that two different key would result in the same hash value.
  - “Collision”
  - Typical hash function:
    - mod by a large prime number
  - Design a way to resolve hash function collision
    - Closed: linear probing
    - Open

Hash table

- Close hash table
  - Linear probing
  - Solution: append the value in the next available spot starting from the desired position.

Hash table



Hash table

- Closed hash table (linear probing)
  - Search:
    - Compute the hash value
    - Linear scan starting from the hash value to an empty slot
    - Nearly O(1), depending on the load factor

Hash table

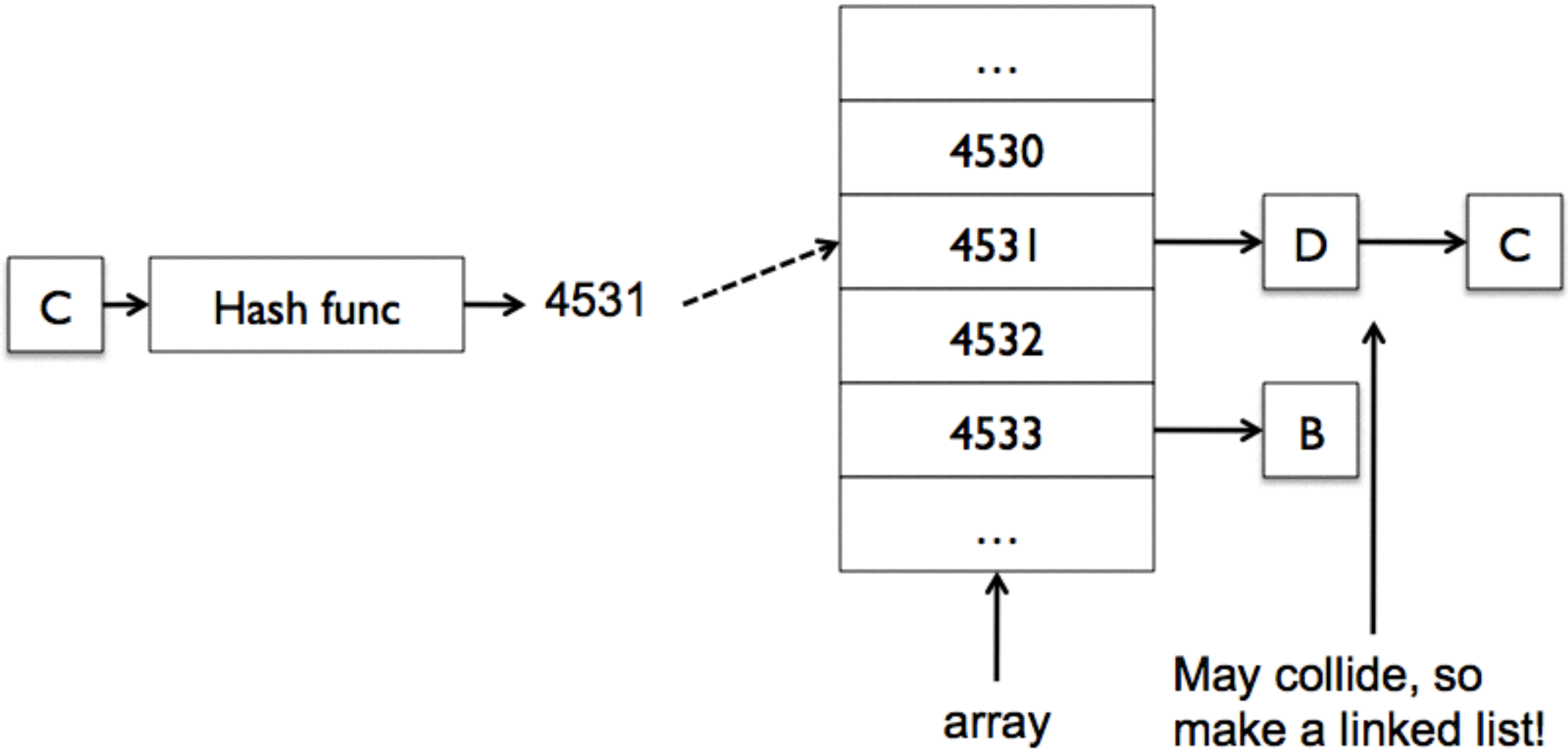
- Closed hash table (linear probing)
  - Problem:
    - Deletion
    - Hard to maintain data integrity for the hash table.

Hash table

- Open hash table
  - Use Linkedlist for each hash value (bucket)
  - Maintain the linkedlist for collision
    - Insert: append a new node
    - Delete: delete a node from the linked list

Hash table

- Open hash table



Hash table

- Open hash table
  - Search:
    - Find the corresponding bucket
    - Traverse the linkedlist to find the item

Hash table

- Complexity analysis
  - Desired performance:
    - Insertion, deletion, search: O(1)
  - Collision ruins the wish
    - Insertion, deletion and search would take longer
    - But approximately O(1)
  - Based on the load factor and how frequent a collision from the hash function happens.
    - Generally open hash table performs better than closed hash table using linear probing.

Hash table

- Compared with Binary Search Tree

	Hash table	Binary Search Tree
Speed	O(1)	O(log n)
Max size	Closed: by array size Open: unlimited	unlimited
Space efficiency	Waste a lot of memory	Only memory needed
Ordering	No ordering (random)	sorted

Hash table

- To keep a better performance, keep a low load factor.
  - A waste of memory.
  - Tradeoffs between space and speed.

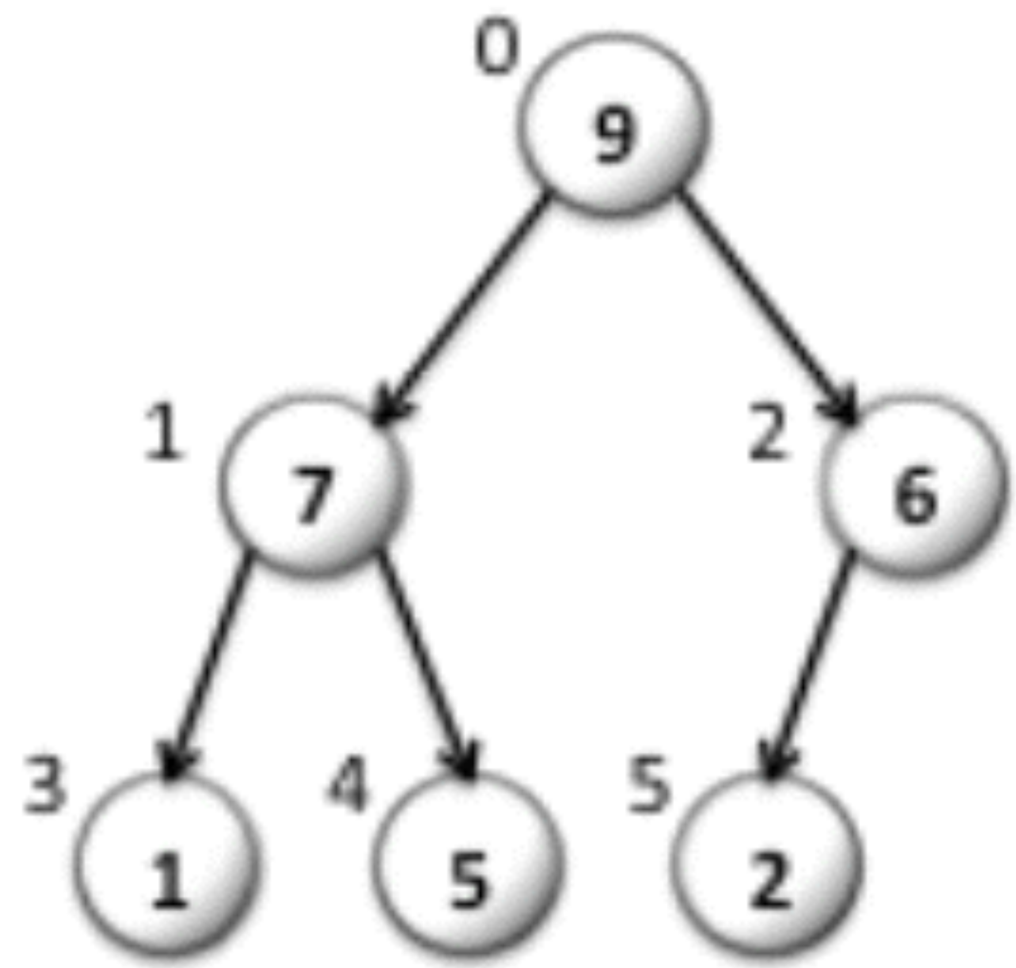
Heap

- ADT having easy access to largest or smallest item in your data
  - Maxheap & minheap
  - Can be used to implement priority queue
  - O(1) for getting the largest/smallest item
  - O(log n) for inserting an item
  - O(log n) for removing largest/smallest item

  - What if we use BST?

Heap

- Heap is a complete binary tree
- Each node's value is larger than or equals to it's children's.
  - Maxheap
  - Every subtree is a maxheap too.



Heap

- Complete Binary Tree
  - L level
  - For first L-1 Levels, tree is full
  - For level L, nodes are on the left

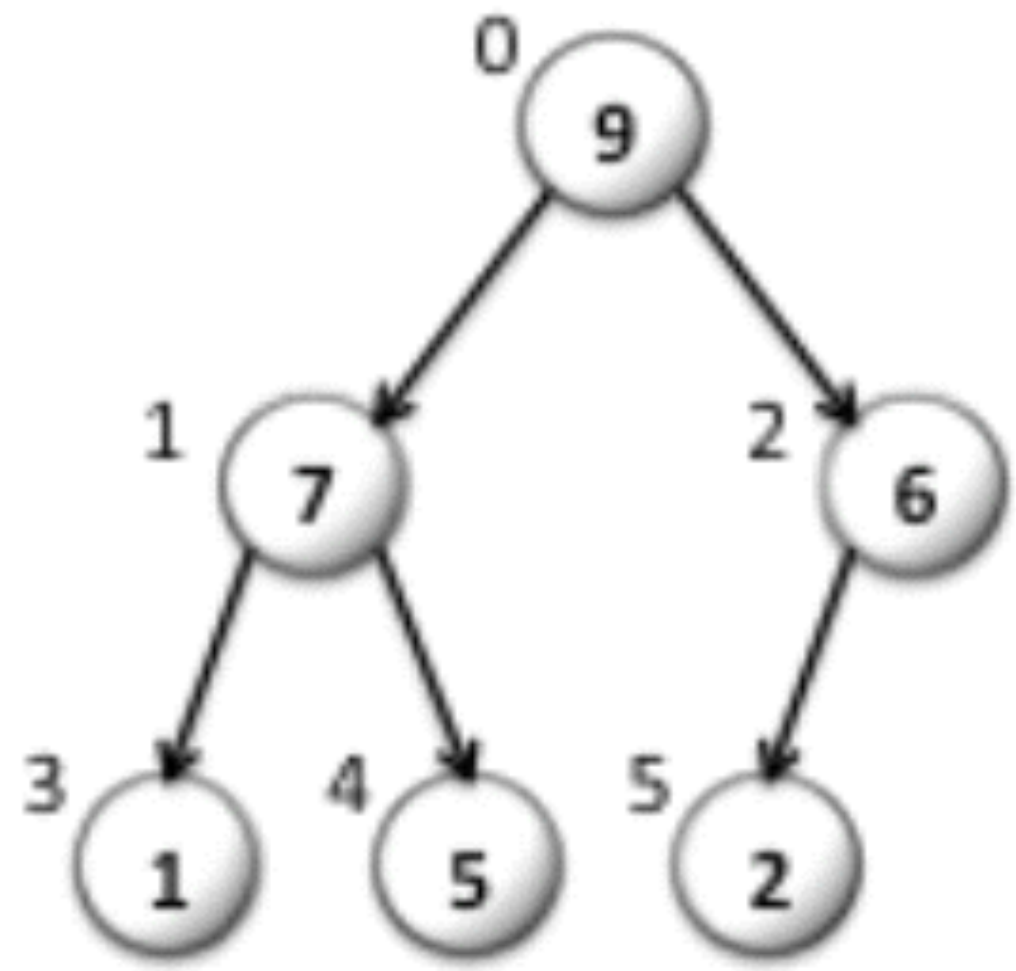
  - If N nodes, what's the level of the tree?
    - Guaranteed log2(N)

Heap

- To maintain largest item on the top
  - How to insert an item
  - How to remove the largest item

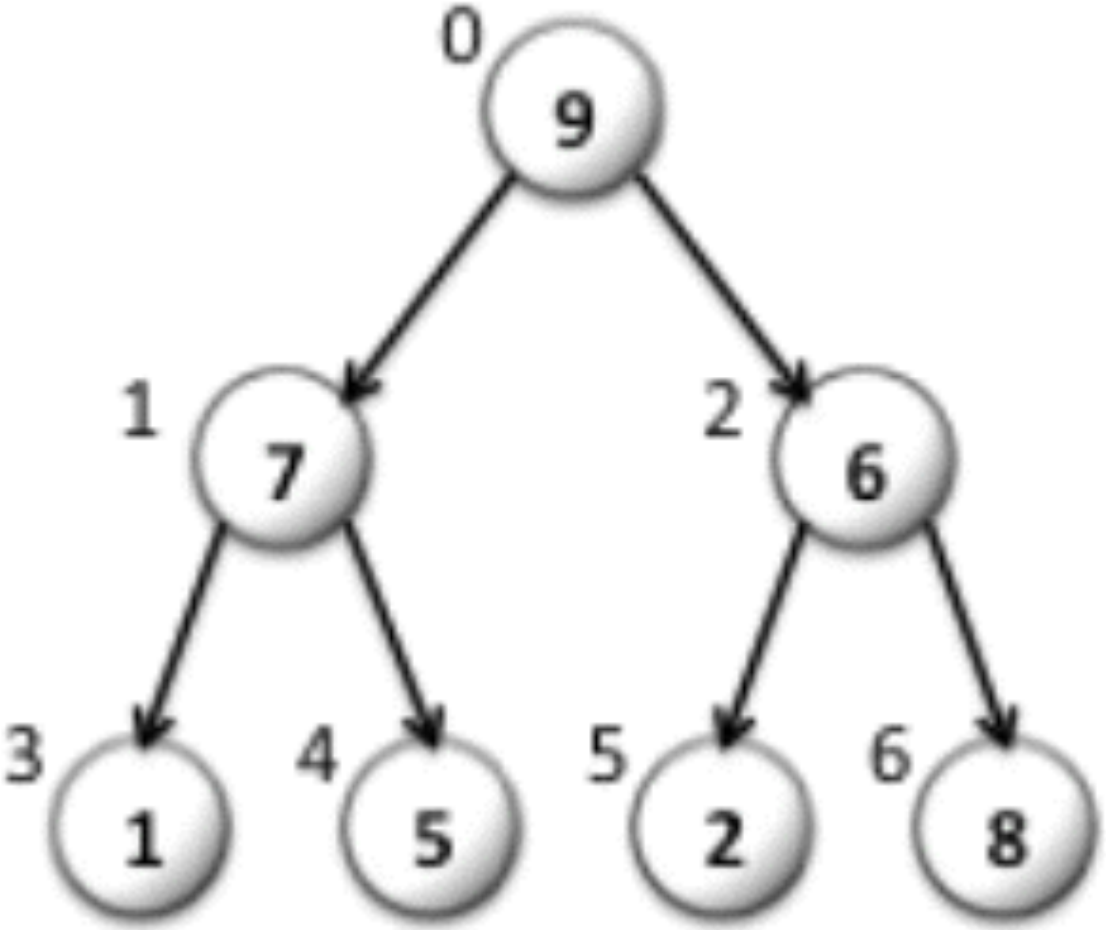
Heap

- Insert an item into maxheap
  - Insert 8
  - Append the next node
  - Bubble up to make it still a maxheap



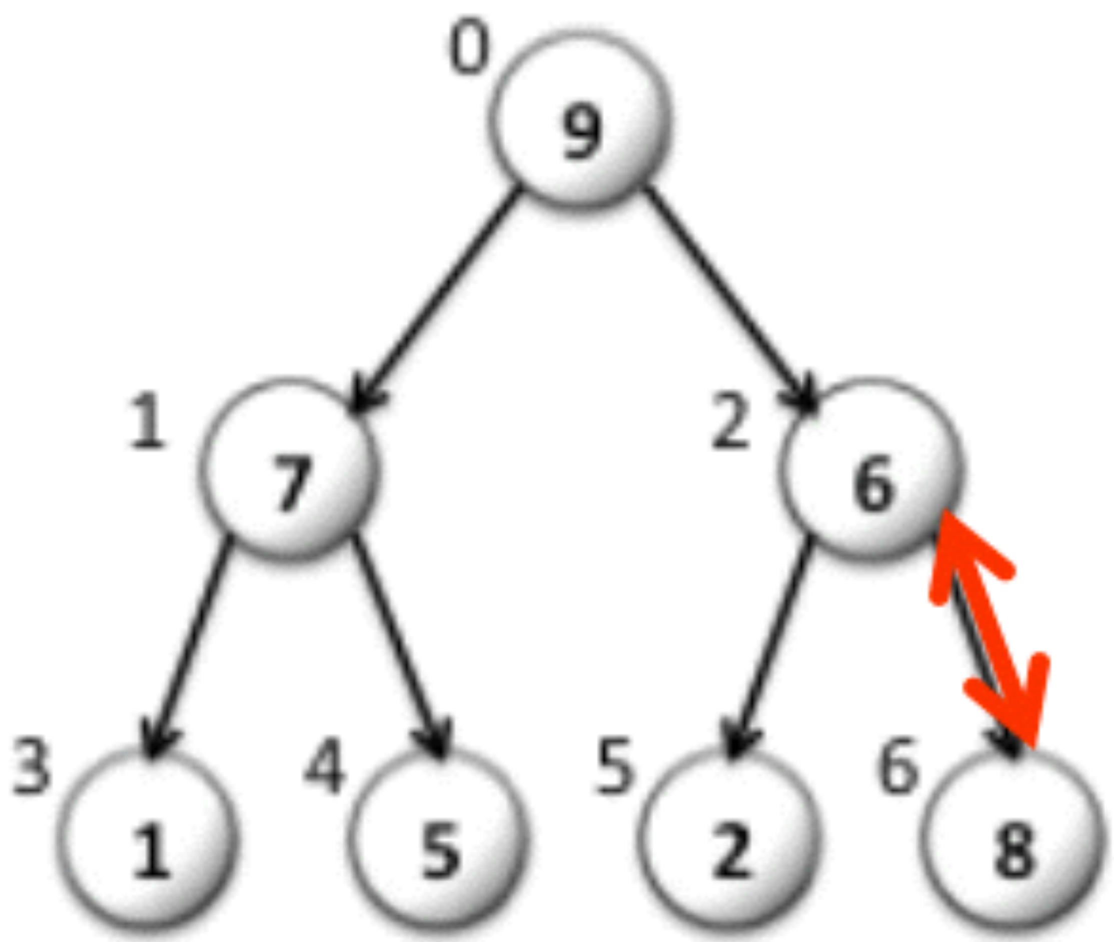
Heap

- Insert an item into maxheap
  - Insert 8
  - Append the next node
  - Bubble up to make it still a maxheap
    - Is it larger than parent?
    - Swap if yes.



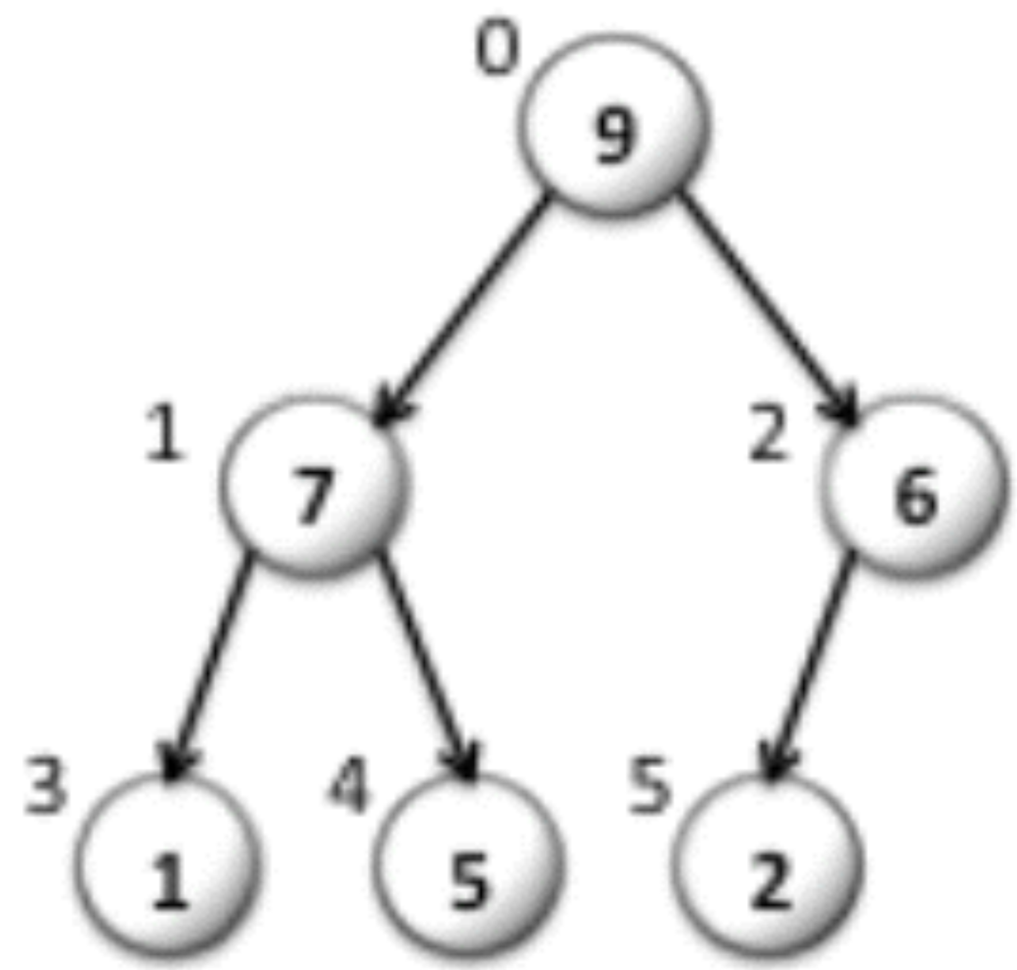
Heap

- Insert an item into maxheap
  - Insert 8
  - Append the next node
  - Bubble up to make it still a maxheap
    - Is it larger than parent?
    - Swap if yes.
  - Complexity?
    - O(log n)



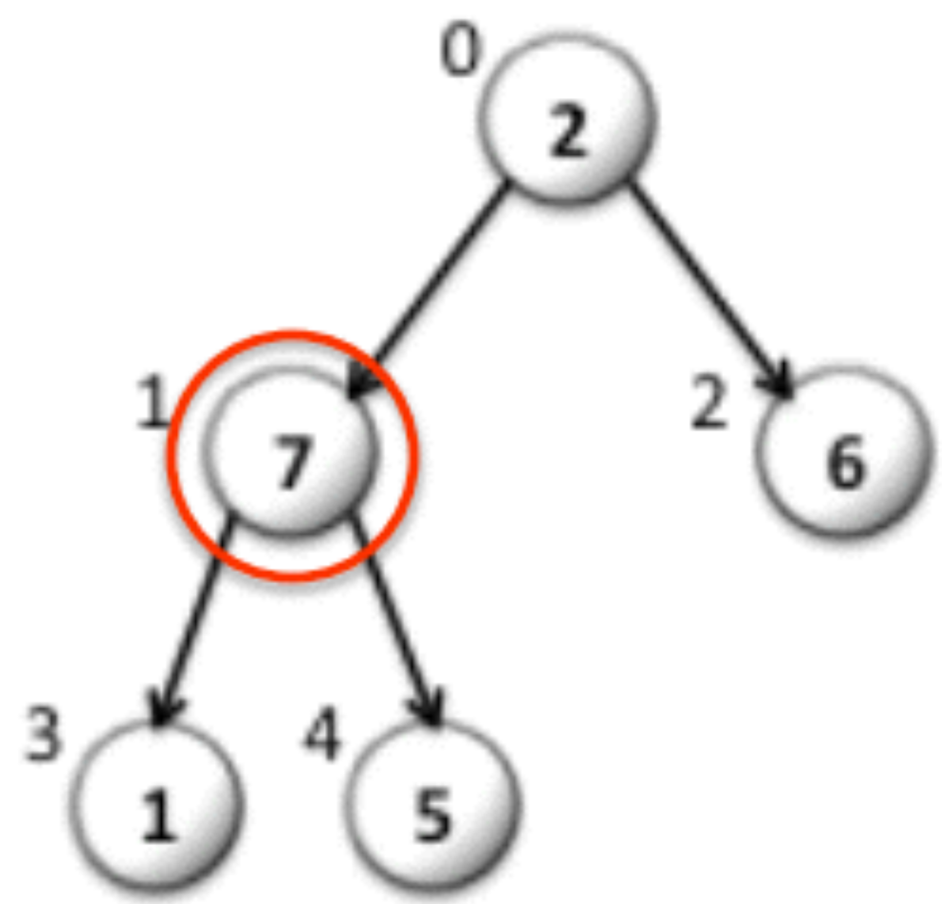
Heap

- Extract largest item from maxheap
  - Need to re-organize the tree to maintain largest one on top.
  - Find the last node, place the value to top node
    - Delete last node then
  - Bubble down, re-organize
    - Compare two children
    - Swap with larger one



Heap

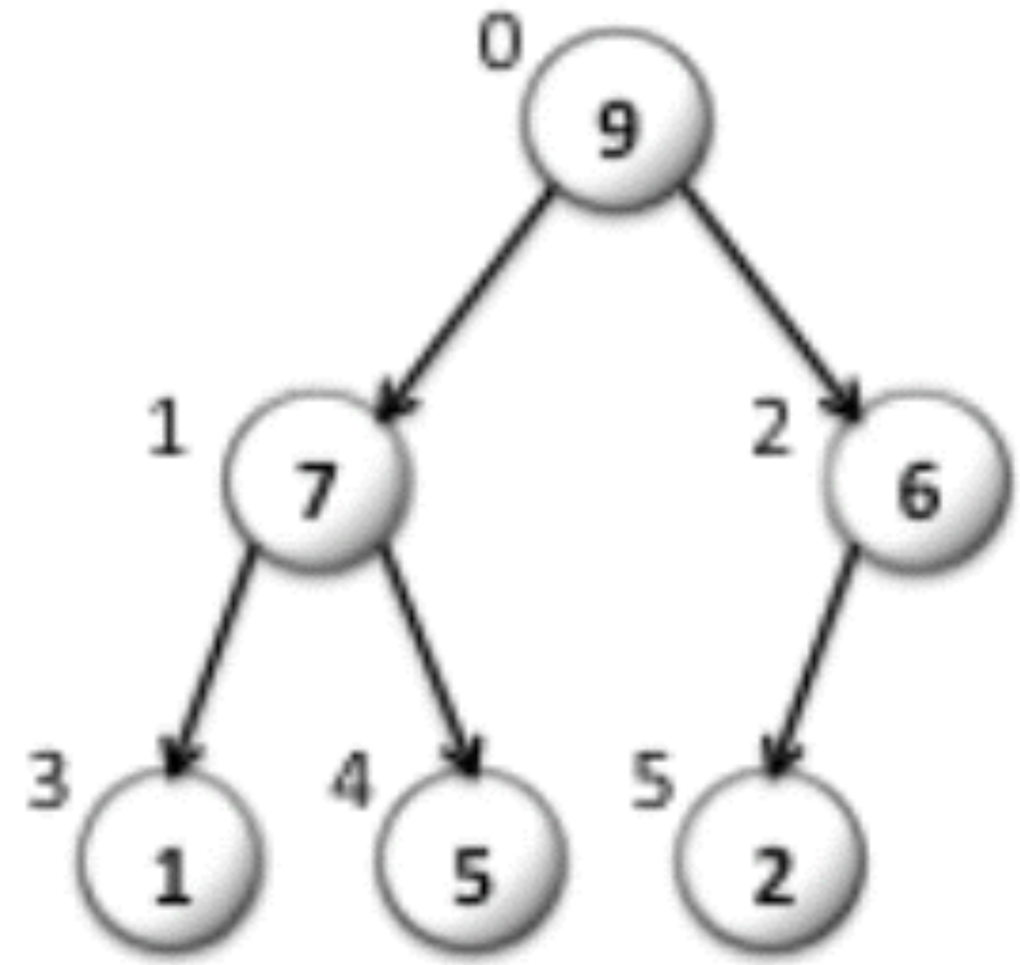
- Extract largest item from maxheap
  - Need to re-organize the tree to maintain largest one on top.
  - Find the last node, place the value to top node
    - Delete last node then
  - Bubble down, re-organize
    - Compare two children
    - Swap with larger one



Heap

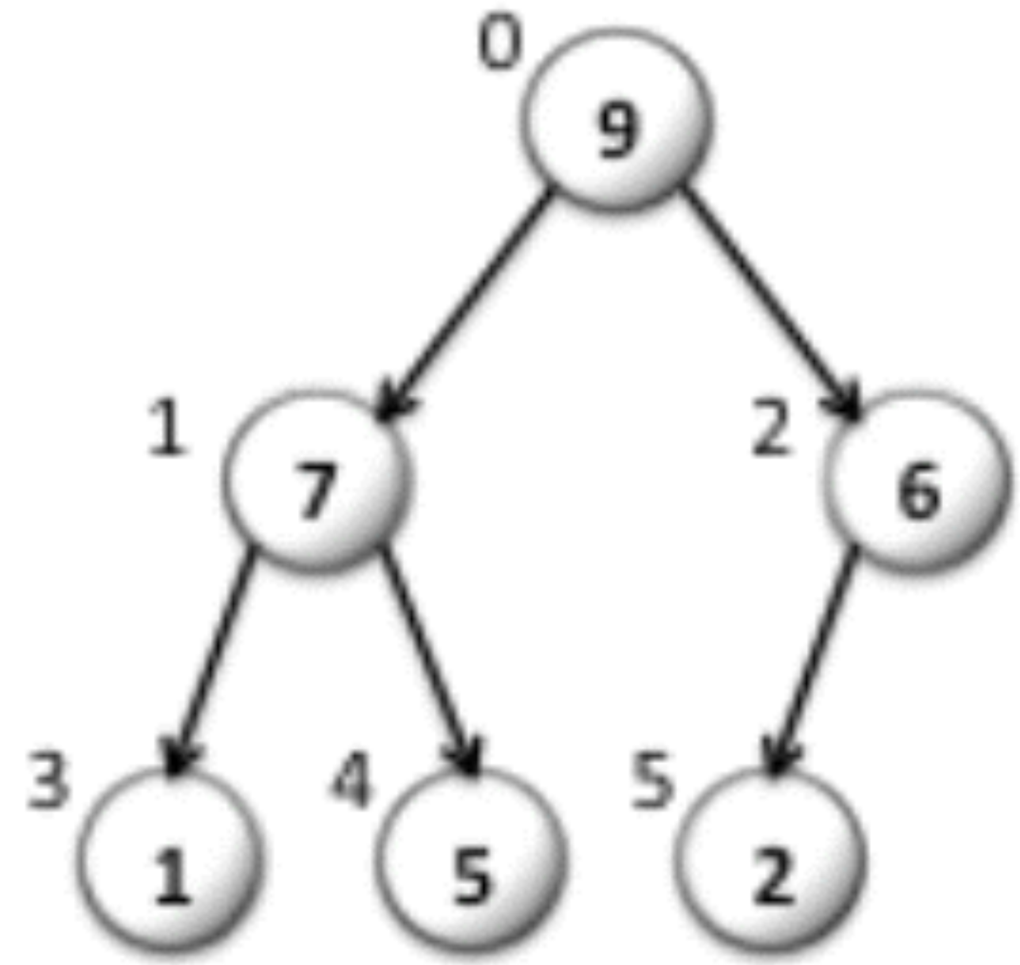
- Heap implementation
  - Using an array
  - Store data in level-ordering order

9	7	6	1	5	2



Heap

- Array Implementation
  - Last node
    - Last position
  - Given a node at position i
    - Parent position?
    - (i – 1) /2
  - Given a node at position i
    - Children position?
    - 2*i + 1, 2*i + 2



9	7	6	1	5	2

Heapsort

- Use heap to sort
  - Insert items in a maxheap
  - Extract largest item from maxheap one by one to get the data sorted

  - 2, 6, 3, 1, 5, 4, 7, 8
  - Show how the heap looks like after each step
    - Insert into a maxheap
    - Extract max from the heap

Heapsort

- In-place heap sort
  - You could build the maxheap in your array, without using another array.
  - Heapification
    - General idea: build maxheap from bottom, swap items if necessary.
    - 2, 6, 3, 1, 5, 4, 7, 8
  - Extracting max is same as before

Heapsort

- Complexity
  - Heapification: O(n)
  - Extracting max:
    - N times
    - Each time O(log n) for maintaining maxheap
    - O(nlog n)
  - Total: O(nlog n)
- Advantage:
  - Guaranteed O(nlog n)
  - In-place sorting