Fall 2019

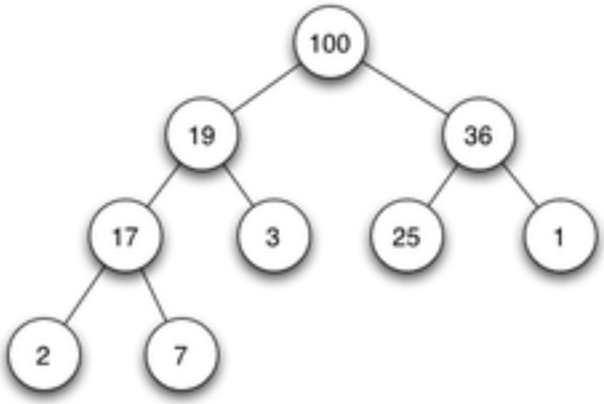
Data Structures Lecture 7

Fang Yu

Software Security Lab.
Dept. Management Information Systems,
National Chengchi University

Recap

- We have talked about object oriented programing
  - Chapter 1, 2, 12
- Basic Data Structures
  - Linked Lists, Arrays, Stacks, Queues
    - Chapter 3, 5, 6
  - Trees and Heaps
    - Chapter 7 and 8

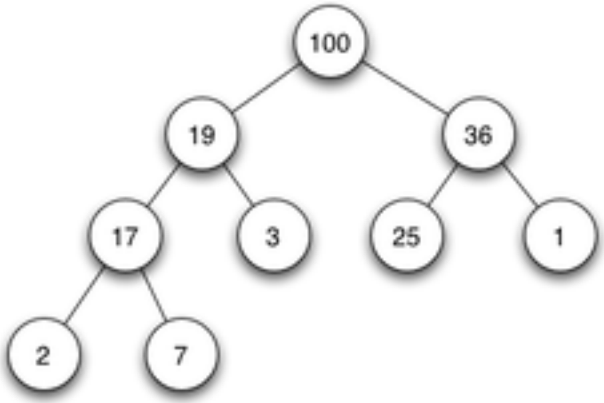


One Kind of Binary Tree ADTs

Heaps and Priority Queues

Heap

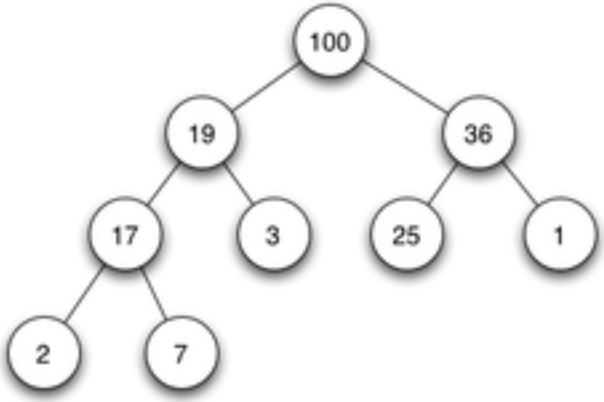
- A binary tree storing keys at its nodes



Heap

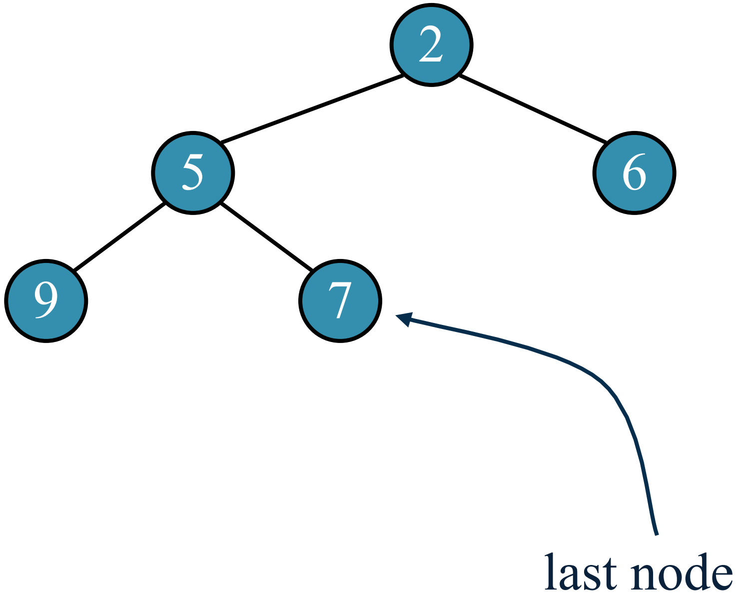
Satisfy the following properties:

- Heap-Order:
  - for every internal node v other than the root,
  - ***Maxheap: key***(***v***) <= ***key***(***parent***(***v***))
  - ***Minheap: key***(***v***) >= ***key***(***parent***(***v***))
- A Complete Binary Tree:
  - let h be the height of the heap
  - for $i = 0, \ldots, h - 1$, there are 2^i nodes of depth i
  - at depth $h - 1$, the internal nodes are to the left of the external nodes



Heap

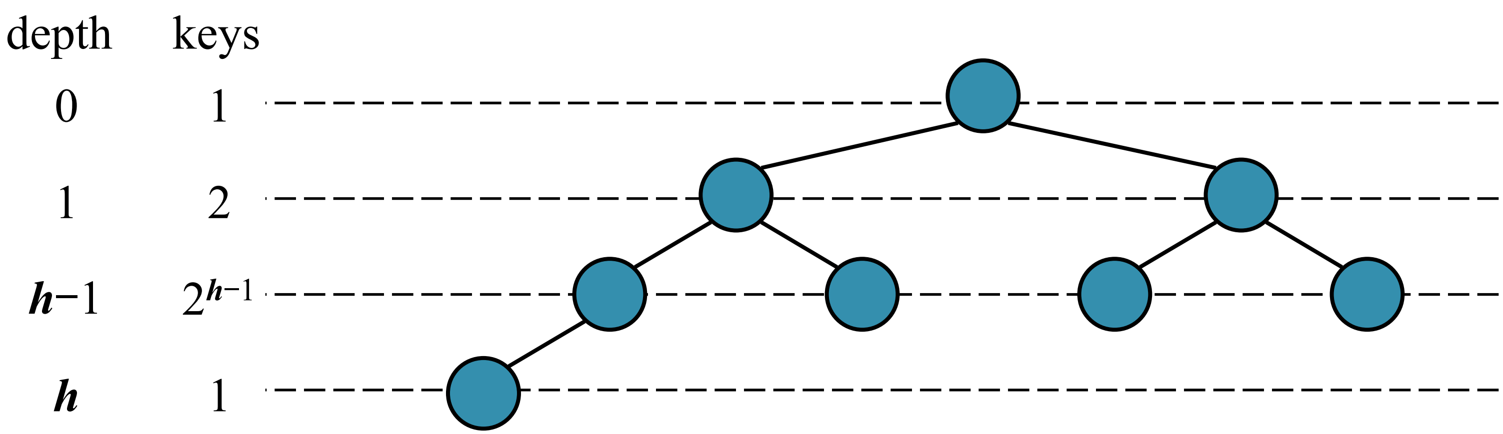
- The last node of a heap is the rightmost node with the maximal depth



Height of a Heap

- Theorem:

  A heap storing $\boldsymbol{n}$ keys has height $\boldsymbol{O}(\log \boldsymbol{n})$



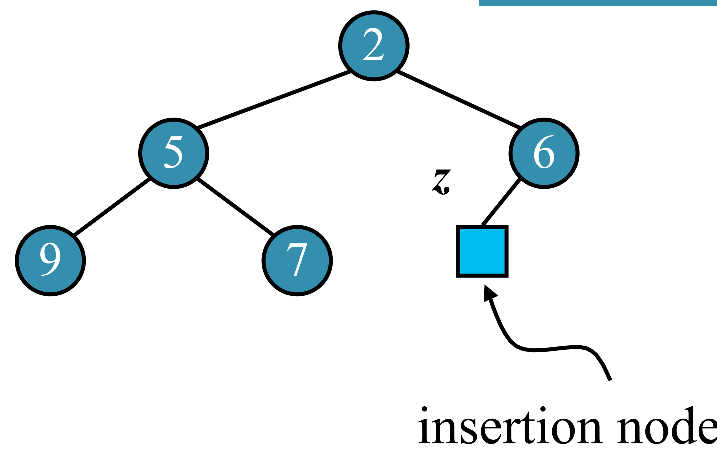
Height of a Heap

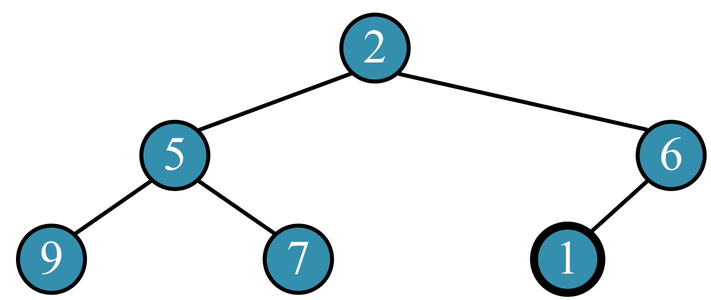
Proof: (we apply the complete binary tree property)

- Let $\boldsymbol{h}$ be the height of a heap storing $\boldsymbol{n}$ keys
- Since there are 2^i keys at depth $\boldsymbol{i} = 0, \ldots, \boldsymbol{h} - 1$ and at least one key at depth $\boldsymbol{h}$, we have $\boldsymbol{n} \geq 1 + 2 + 4 + \ldots + 2^{\boldsymbol{h}-1} + 1$
- Thus, $\boldsymbol{n} \geq 2^{\boldsymbol{h}}$, i.e., $\boldsymbol{h} \leq \log \boldsymbol{n}$

Insertion

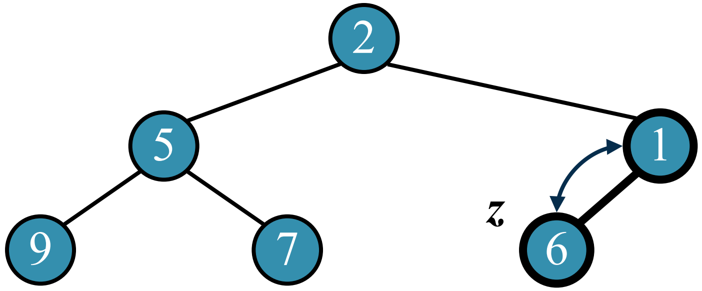
- Insert a key ***k*** to the heap
  - a complete binary tree
  - heap order
- The algorithm consists of three steps
  - Find the insertion node ***z*** (the new last node)
  - Store ***k*** at ***z***
  - Restore the heap-order property (discussed next)

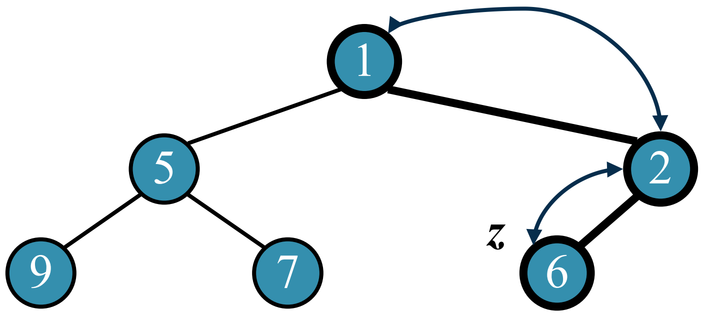




Upheap

- After the insertion of a new key ***k***, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping ***k*** along an upward path from the insertion node



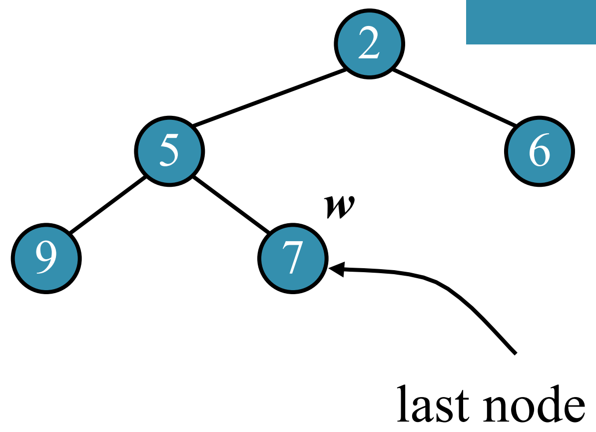


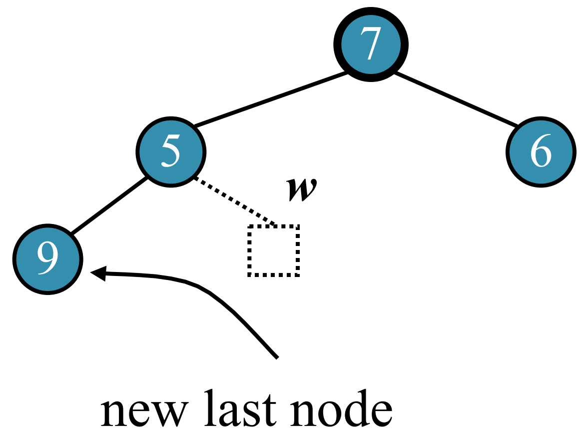
Upheap

- Upheap terminates when the key ***k*** reaches the root or a node whose parent has a key smaller than or equal to ***k***
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time
- Insertion of a heap runs in $O(\log n)$ time

RemoveMin

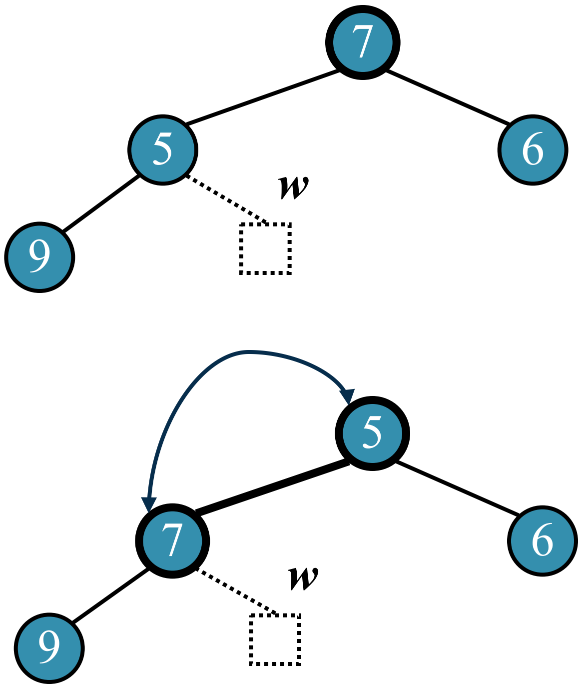
- Removal of **the root** key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node ***w***
  - Remove ***w***
  - Restore the heap-order property (discussed next)





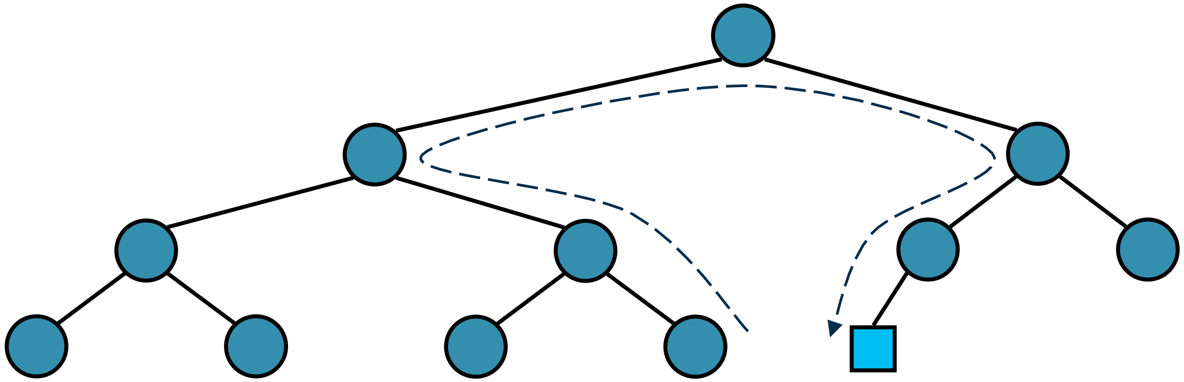
Downheap

- After replacing the root key with the key ***k*** of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key ***k*** along a downward path from the root
  - Find the minimal child c
  - Swap k and c if c<k



Updating the Last Node

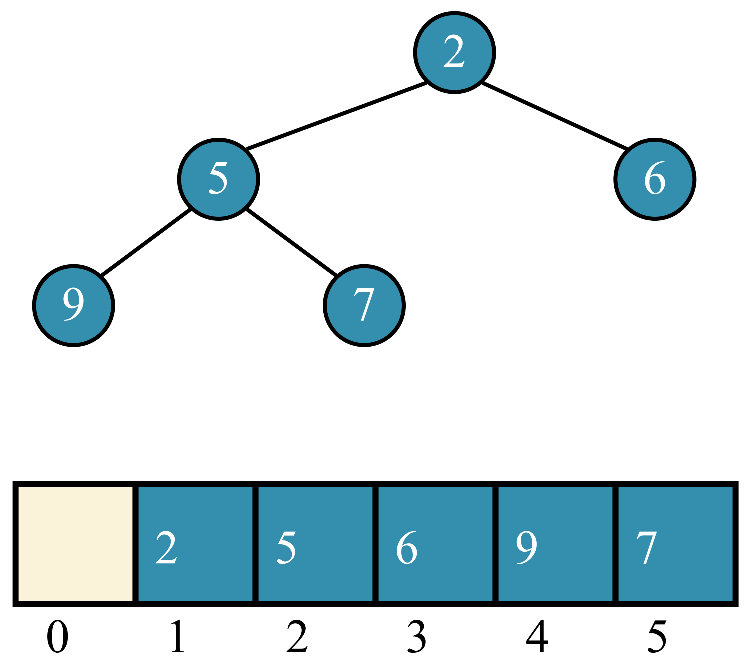
- The insertion node can be found by traversing a path of $\boldsymbol{O}(\log \boldsymbol{n})$ nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until a leaf is reached



- Similar algorithm (swap left/right) for updating the last node after a removal

Array-based Implementation

- We can represent a heap with n keys by means of an array of length $n + 1$
- The cell of at rank 0 is not used
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Insert at rank $n + 1$
- Remove at rank n
- Use a ***growthable array***



Recall: Priority Queue ADT

- A priority queue dequeues entries in order according to their keys
- Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, x)
    inserts an entry with key k and value x
  - removeMin()
    removes and returns the entry with smallest key
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), isEmpty()

Sequence-based Priority Queue

- Implementation with an unsorted list

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

Sequence-based Priority Queue

- Implementation with a sorted list

- Performance:
  - insert takes $\boldsymbol{O}(\boldsymbol{n})$ time since we have to find the place where to insert the item
  - removeMin and min take $\boldsymbol{O}(1)$ time, since the smallest key is at the beginning

Priority Queue Sort

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm ***PQ-Sort*(*S*, *C*)**
 Input sequence ***S***, comparator ***C*** for the elements of ***S***
 Output sequence ***S*** sorted in increasing order according to ***C***
 P ← priority queue with comparator ***C***
 while ¬***S.isEmpty*** ()
  e ← ***S.removeFirst*** ()
  P.insert (***e***, ∅)
 while ¬***P.isEmpty***()
  e ← ***P.removeMin***().***getKey***()
  S.addLast(***e***)

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence

- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

- Selection-sort runs in $O(n^2)$ time

Selection-Sort Example

	Sequence S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	.. ..	
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence

- Running time of Insertion-sort:

1. Inserting the elements into the priority queue with n insert operations takes time proportional to

$$1 + 2 + \ldots + n$$

2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time

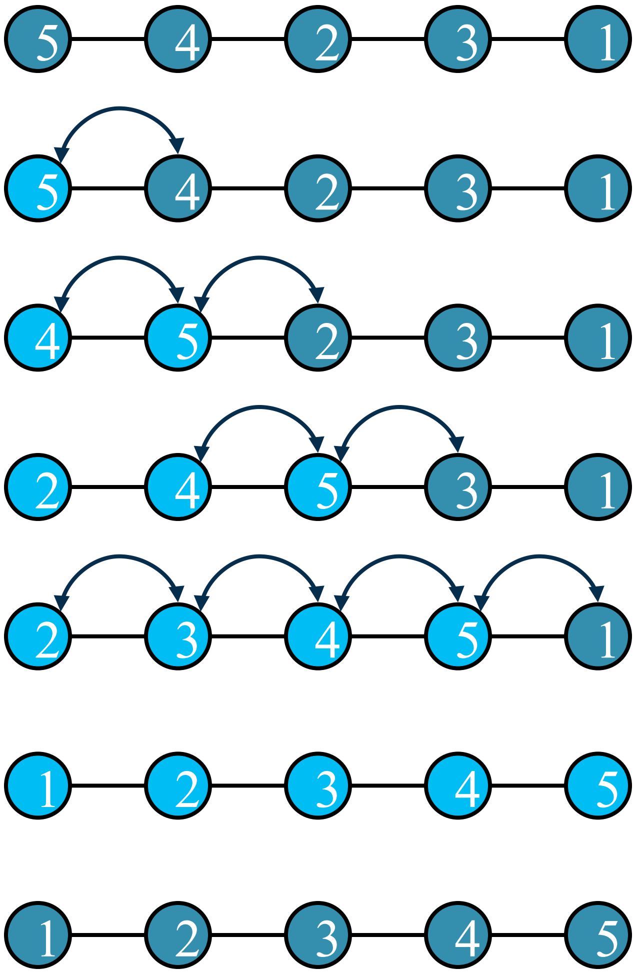
- Insertion-sort runs in $O(n^2)$ time

Insertion-Sort Example

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

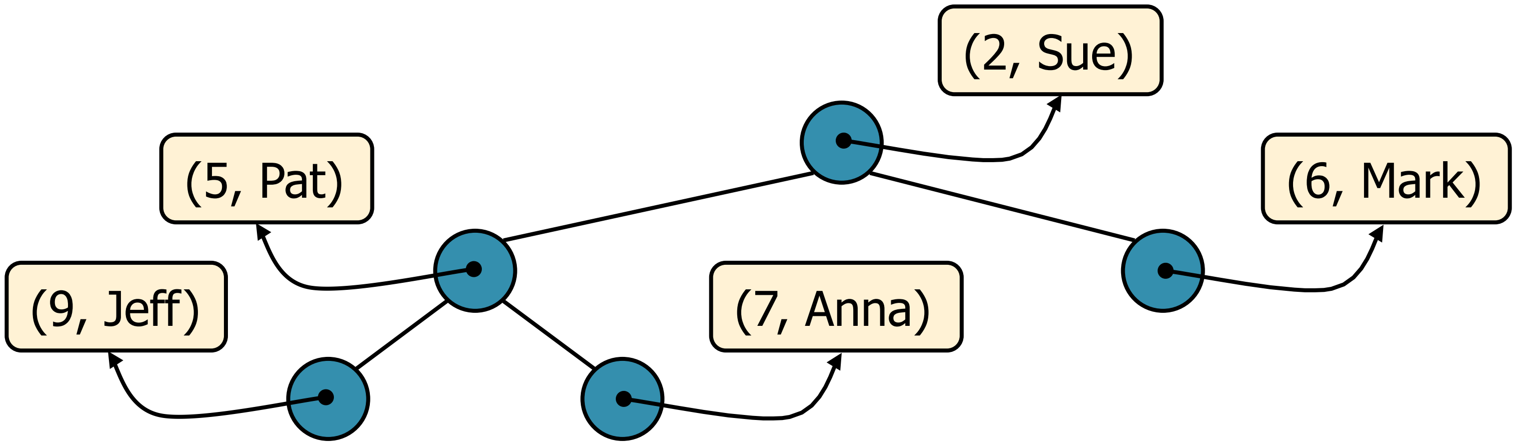
In-place Insertion-Sort (Bubble Sort)

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instead of modifying the sequence



Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node



© 2010 Goodrich, Tamassia

Heap-Sort

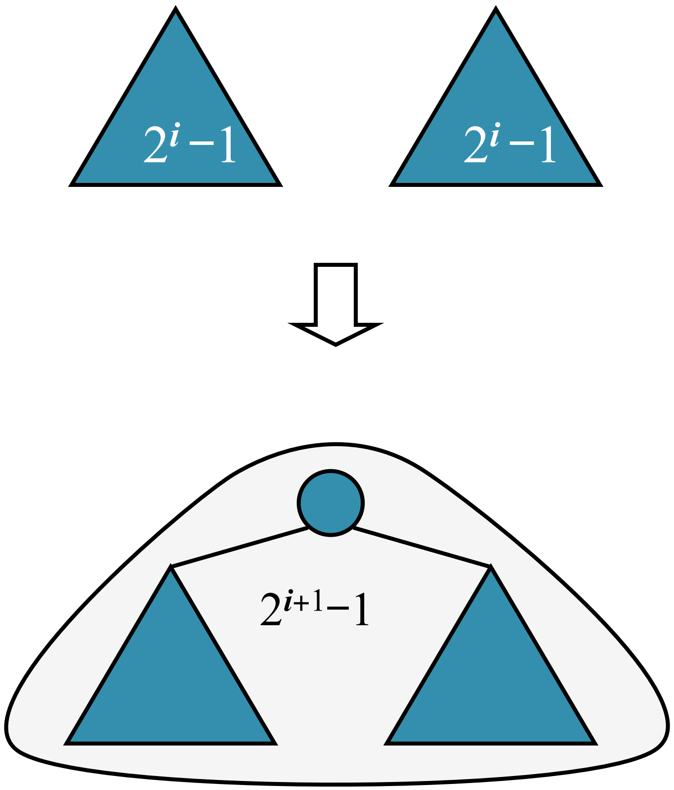
- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insert and removeMin take $O(\log n)$ time
  - methods size, isEmpty, and min take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

A Faster Heap-Sort

- Insert n keys one by one taking O(n log n) times
- If we know all keys in advance, we can save the construction to O(n) times by bottom up construction

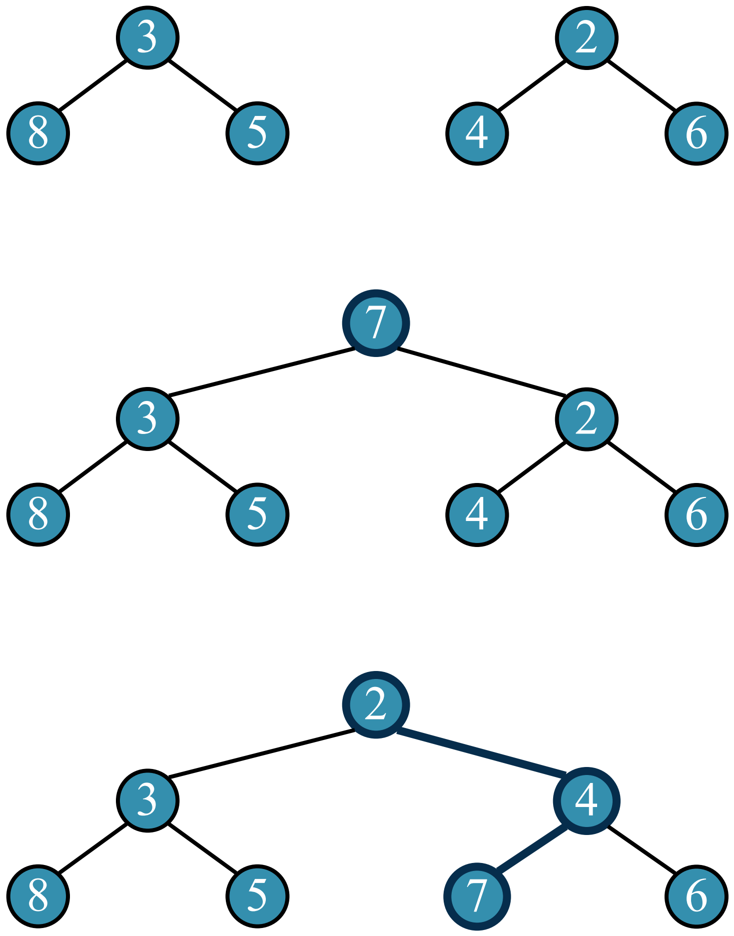
Bottom-up Heap Construction

- We can construct a heap storing n given keys in using a bottom-up construction with log n phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1}-1$ keys

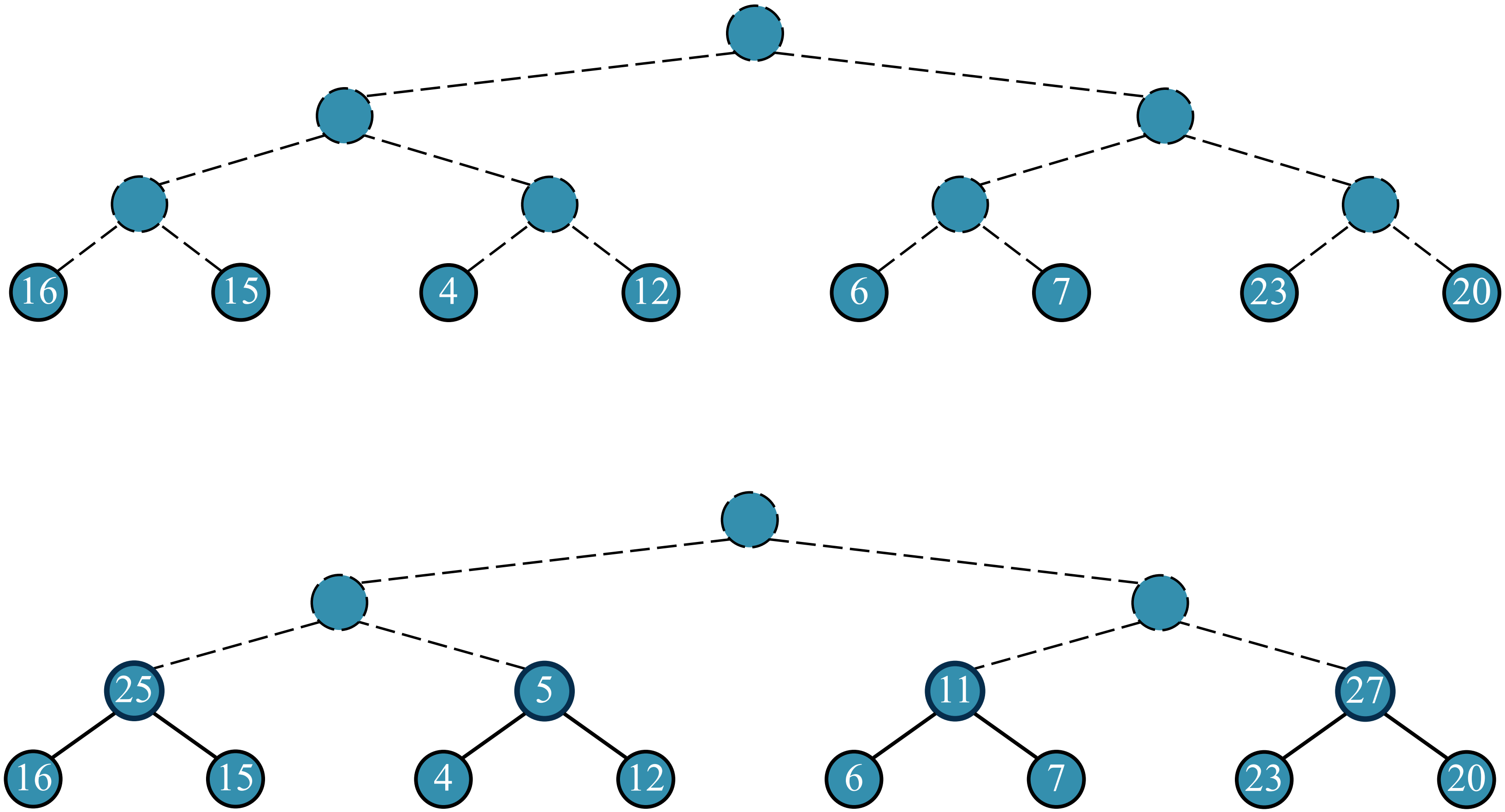


Merging Two Heaps

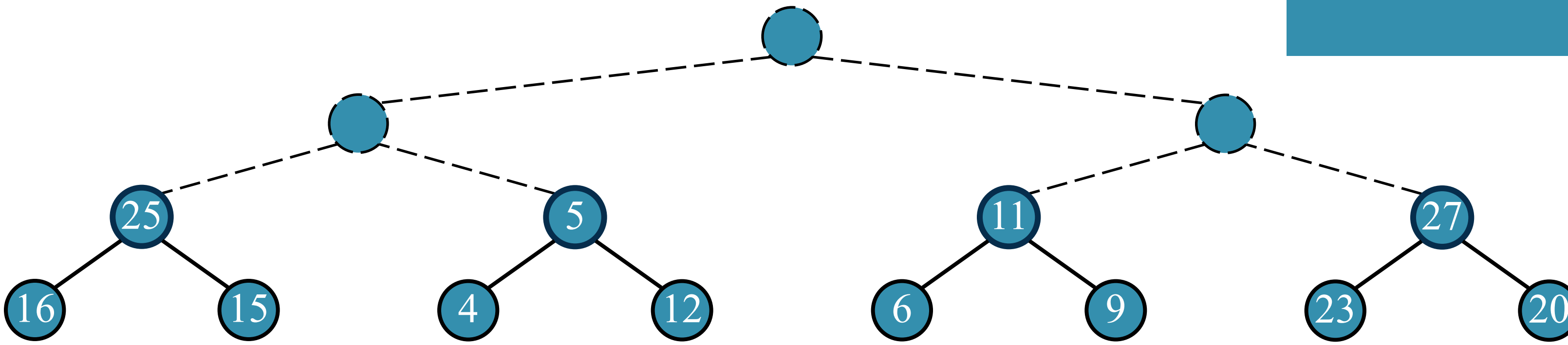
- Given two heaps and a key ***k***, we create a new heap with the root node storing ***k*** and with the two heaps as subtrees
- We perform downheap to restore the heap-order property

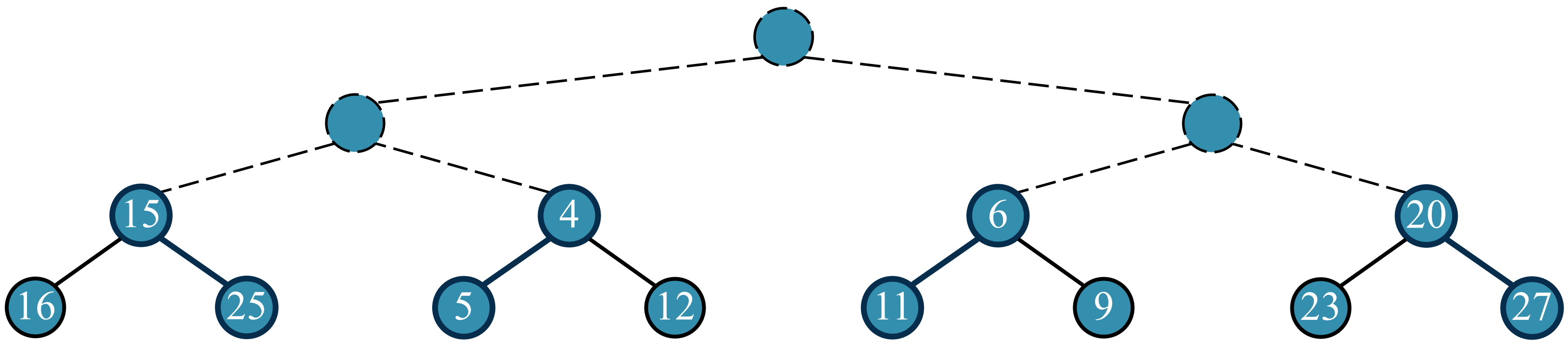


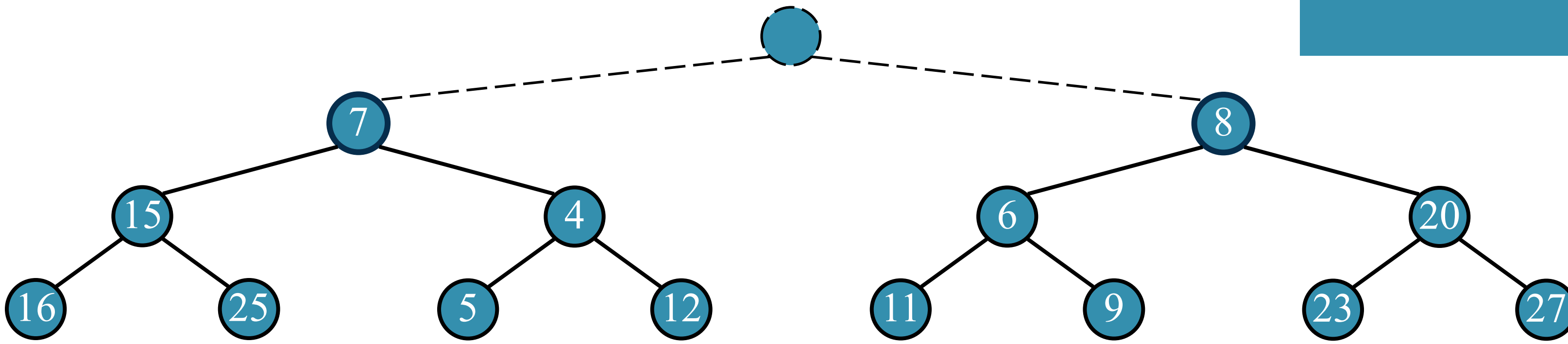
An Example of Bottom-up Construction

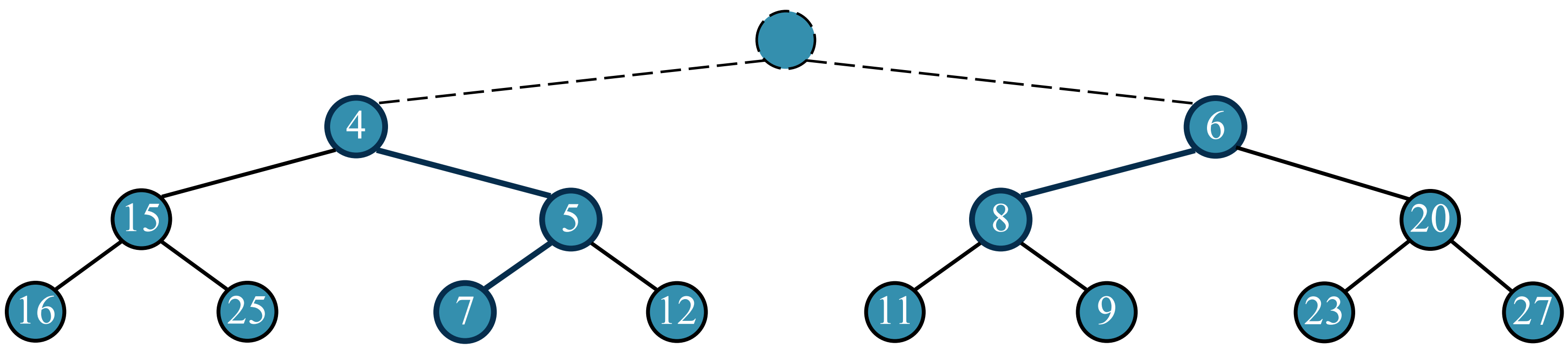


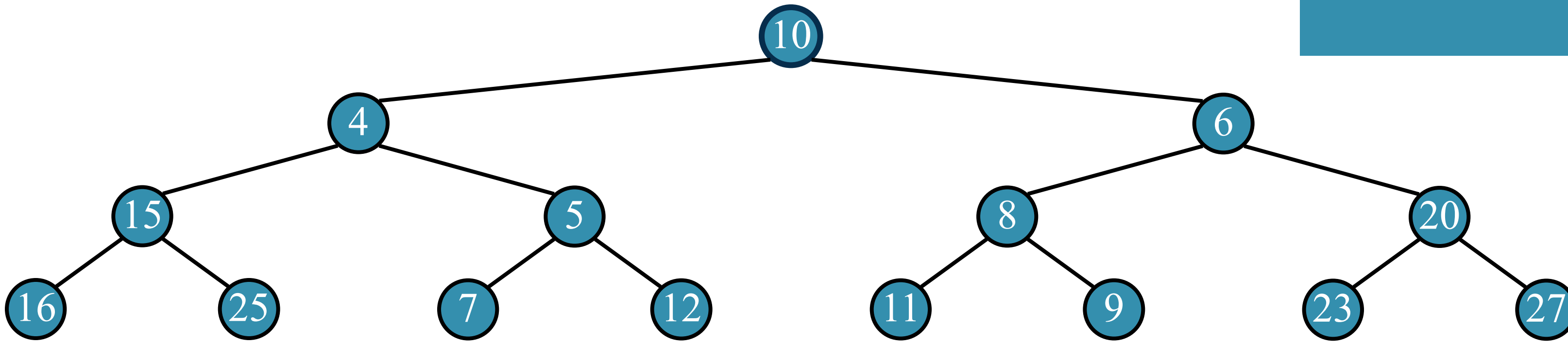
Restore the order for each one

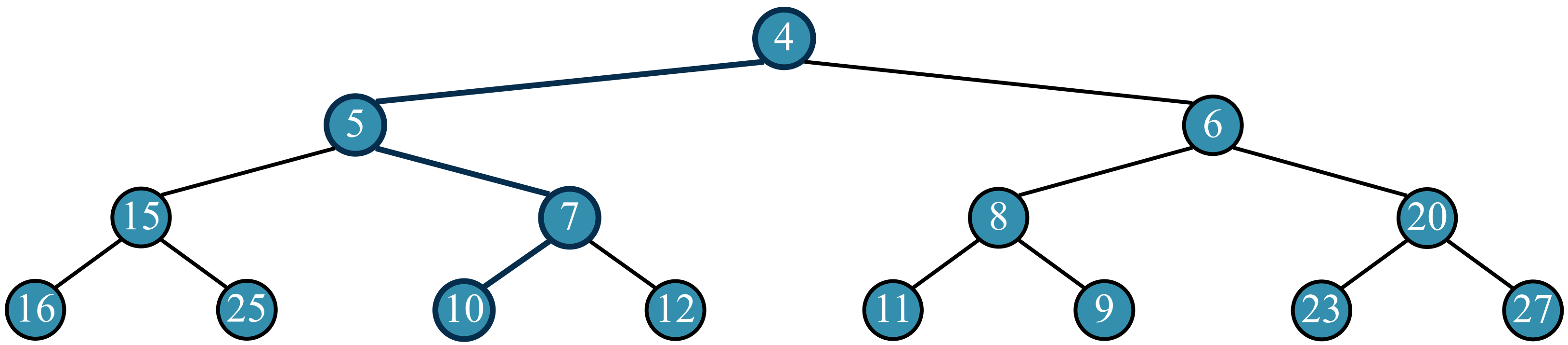





7
8
15
4
6
20
16
25
5
12
11
9
23
27

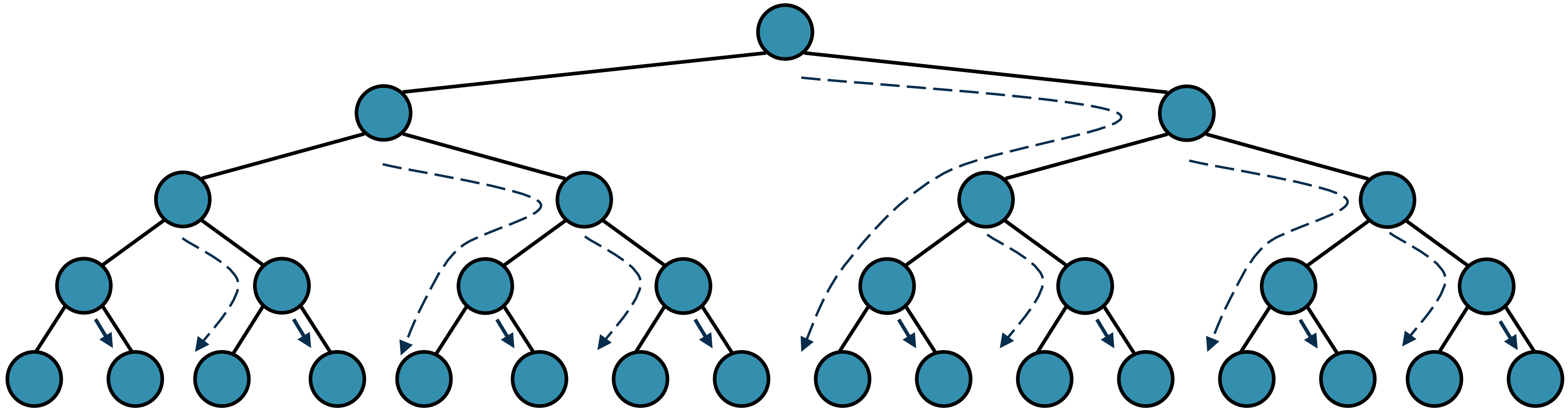

4
6
15
5
8
20
16
25
7
12
11
9
23
27


10
4
6
15
5
8
20
16
25
7
12
11
9
23
27


4
5
6
15
7
8
20
16
25
10
12
11
9
23
27

Analysis

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)



Analysis

- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort from $O(n \log n)$ to $O(n)$

HW7 (Due on 11/21)

Maintain a keyword heap.

- A keyword is a triple [String name, Integer count, Double weight]
- Heap Order: n.count >= n.parent.count
- Use java.util.PriorityQueue
  - http://download.oracle.com/javase/1.5.0/docs/api/java/util/PriorityQueue.html
- Here's an example of a priority queue sorting by string length
- Reuse your code in HW4

Java.util.PriorityQueue

```
// Test.java
import java.util.Comparator;
import java.util.PriorityQueue;
public class Test{
	public static void main(String[] args){
		Comparator<String> comparator = new StringLengthComparator();
		PriorityQueue<String> queue =
		new PriorityQueue<String>(10, comparator);
		queue.add("short");
		queue.add("very long indeed");
		queue.add("medium");
		while (queue.size() != 0) {
			System.out.println(queue.remove());
		 }
	}
}
```

Comparator

```
// StringLengthComparator.java
import java.util.Comparator;
public class StringLengthComparator implements Comparator<String>{
   public int compare(String x, String y)    {
        // Assume neither string is null. Real code should
        // probably be more robust
        if (x.length() < y.length())
            return -1;
        if (x.length() > y.length())
            return 1;
        return 0;
   }
}
```

Operations

Given a sequence of operations in a txt file, parse the txt file and execute each operation accordingly

operations	description
add(Keyword k)	Insert a keyword k to the heap (use offer())
peek()	Output the keyword with the minimal count (use peek())
removeMin()	Return and remove the keyword of the root (the one with the minimal count) (use poll())
output()	Output all keywords in order

An input file

Similar to HW4,

1. You need to read the sequence of operations from a txt file
2. The format is firm
3. Raise an exception if the input does not match the format

```
add Fang 3 1.2
add Yu 5 1.8
add NCCU 2 0.6
add UCSB 11.9
peek
add MIS 4 2.2
removeMin
add Badminton 1 0.6
output
```

[UCSB, 1]
[UCSB, 1]
[Badminton, 1][NCCU, 2][Fang, 3][MIS, 4] [Yu, 5]

Coming Up

- Read Text book Chapter 8
- We will start to talk about algorithms (Chapter 4 and 11) on Nov. 21.