COMP2823

Lecture 5: Priority Queues [GT 5]

Joachim Gudmundsson
School of Computer Science

Some content is taken from material provided by the textbook publisher Wiley.

THE UNIVERSITY OF SYDNEY

Priority Queue ADT

Special type of ADT map to store a collection of key-value items where we can only remove smallest key:

- insert(k, v): insert item with key k and value v
- remove_min(): remove and return the item with smallest key
- min(): return item with smallest key
- size(): return how many items are stored
- is_empty(): test if queue is empty

We can also have a max version of this min version, but we need one structure for each version.

Example

A sequence of priority queue methods:

Method	**Return value**	**Priority queue**
insert(5,A)		{(5,A)}
insert(9,C)		{(5,A),(9,C)}
insert(3,B)		{(3,B),(5,A),(9,C)}
min()	(3,B)	{(3,B),(5,A),(9,C)}
remove_min()	(3,B)	{(5,A),(9,C)}
insert(7,D)		{(5,A),(7,D),(9,C)}
remove_min()	(5,A)	{(7,D),(9,C)}
remove_min()	(7,D)	{(9,C)}
remove_min()	(9,C)	{}
is_empty()	true	{}

Application: Stock Matching Engines

At the heart of modern stock trading systems are highly reliable systems known as **matching engines,** which match the stock trades of buyers and sellers.

Buyers post bids to buy a number of shares of a given stock at or below a specified price

Sellers post offers (asks) to sell a number of shares of a given stock at or above a specified price.

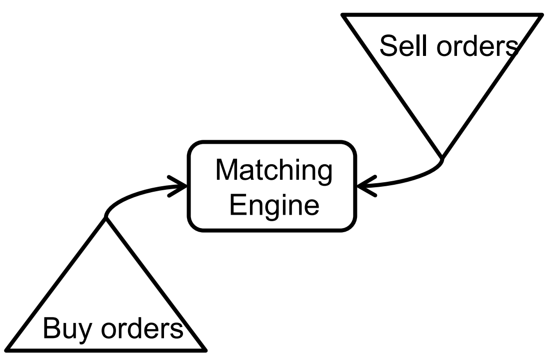
STOCK: EXAMPLE.COM

Buy Orders			Sell Orders		
Shares	Price	Time	Shares	Price	Time
1000	4.05	20 s	500	4.06	13 s
100	4.05	6 s	2000	4.07	46 s
2100	4.03	20 s	400	4.07	22 s
1000	4.02	3 s	3000	4.10	54 s
2500	4.01	81 s	500	4.12	2 s
			3000	4.20	58 s
			800	4.25	33 s
			100	4.50	92 s

Application: Stock Matching Engines

Buy and sell orders are organized according to a price-time priority, where price has highest priority and time is used to break ties

When a new order is entered, the matching engine determines if a trade can be immediately executed and if so, then it performs the appropriate matches according to price-time priority.



STOCK: EXAMPLE.COM

Buy Orders			Sell Orders		
Shares	**Price**	**Time**	**Shares**	**Price**	**Time**
1000	4.05	20 s	500	4.06	13 s
100	4.05	6 s	2000	4.07	46 s
2100	4.03	20 s	400	4.07	22 s
1000	4.02	3 s	3000	4.10	54 s
2500	4.01	81 s	500	4.12	2 s
			3000	4.20	58 s
			800	4.25	33 s
			100	4.50	92 s

Application: Stock Matching Engines

A matching engine can be implemented with two **priority queues**, one for buy orders and one for sell orders.

This data structure performs element removals based on priorities assigned to elements when they are inserted.

```
while True:
    bid ← buy_orders.max()
    ask ← sell_orders.min()
    if bid.price ≥ ask.price then
        bid ← buy_orders.remove_max()
        ask ← sell_orders.remove_min()
        carry out trade (bid, ask)
```

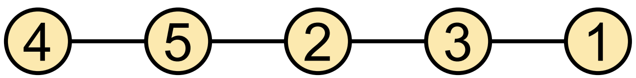
STOCK: EXAMPLE.COM

Buy Orders			Sell Orders		
Shares	**Price**	**Time**	**Shares**	**Price**	**Time**
1000	4.05	20 s	500	4.06	13 s
100	4.05	6 s	2000	4.07	46 s
2100	4.03	20 s	400	4.07	22 s
1000	4.02	3 s	3000	4.10	54 s
2500	4.01	81 s	500	4.12	2 s
			3000	4.20	58 s
			800	4.25	33 s
			100	4.50	92 s

Other applications: sorting, shortest path, task scheduling, data compression, routing, and many more.

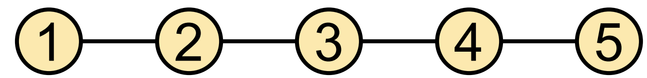
Sequence-based Priority Queue

Unsorted list implementation



– insert in O(1) time since we can insert the item at the beginning or end of the sequence
– remove_min and min in O(n) time since we have to traverse the entire list to find the smallest key

Sorted list implementation



– insert in O(n) time since we have to find the place where to insert the item
– remove_min and min in O(1) time since the smallest key is at the beginning

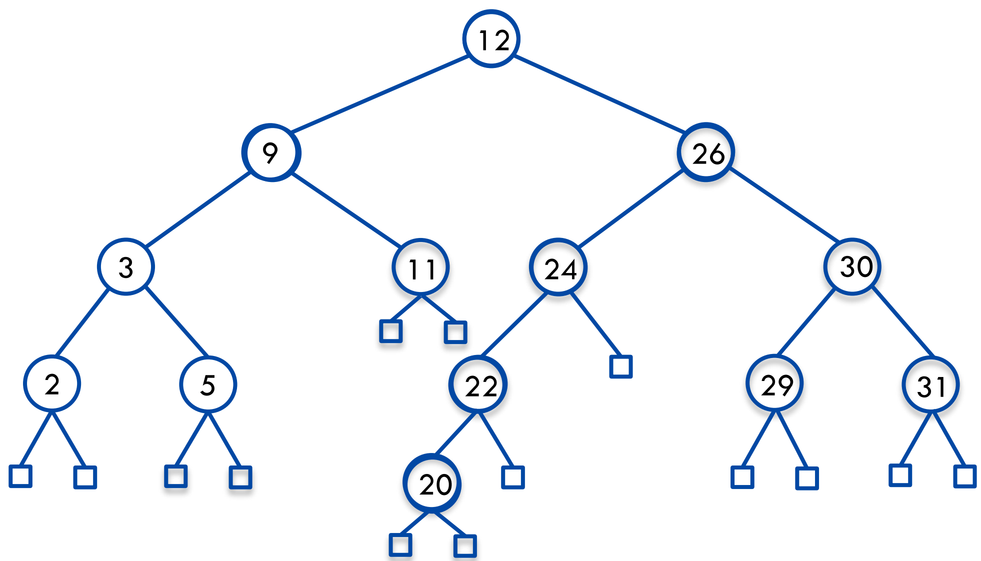
Method	Unsorted List	Sorted List
size, isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

BST-based Priority Queue

– insert in O(log n) time

– remove_min and min in O(log n) time

Method	Unsorted List	Sorted List	BST
size, isEmpty	$O(1)$	$O(1)$	O(1)
insert	$O(1)$	$O(n)$	O(log n)
min, removeMin	$O(n)$	$O(1)$	O(log n)

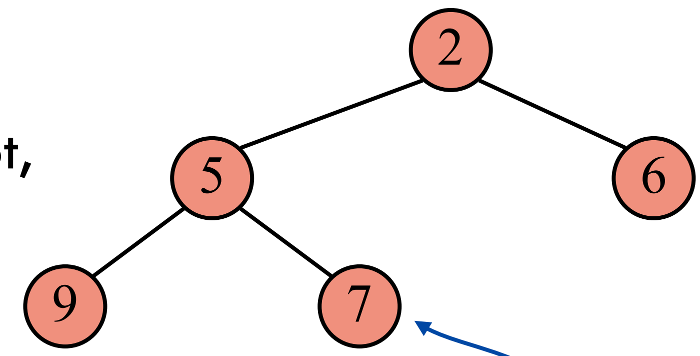


Heap data structure (min-heap)

A heap is a binary tree storing (key, value) items at its nodes, satisfying the following properties:

1. Heap-Order: for every node $v \neq$ root,

   $\text{key}(m) \geq \text{key}(\text{parent}(v))$

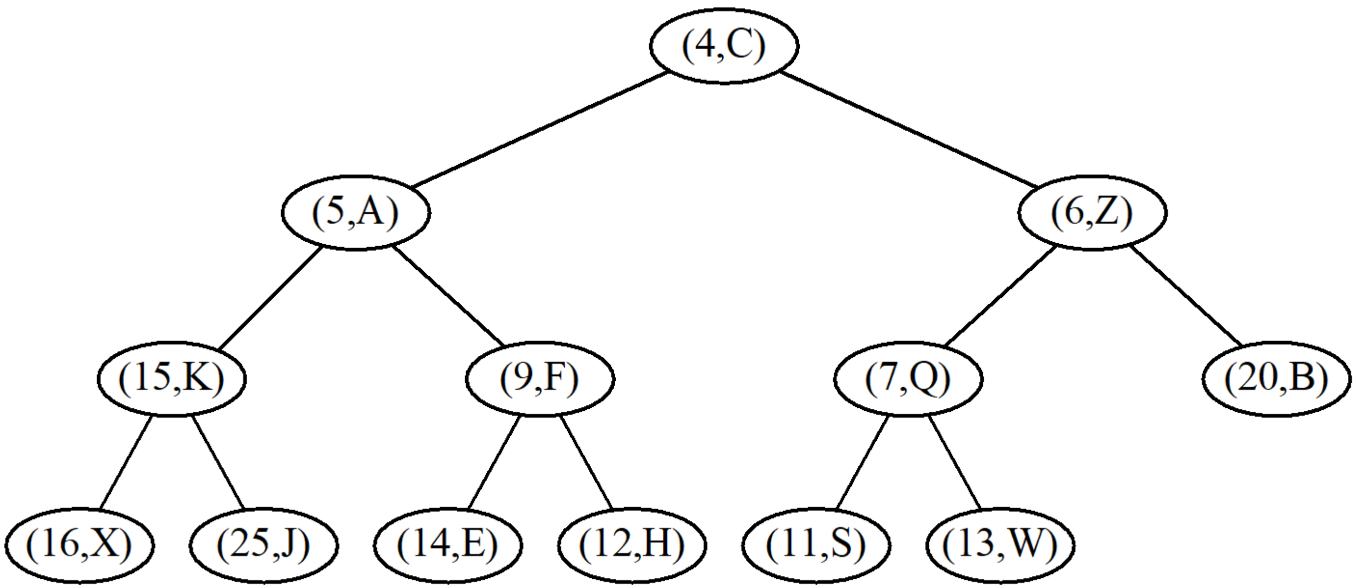


2. Complete Binary Tree: let h be the height
   – every level $i < h$ is full (i.e., there are 2^i nodes)
   – remaining nodes take leftmost positions of level h

The last node is the rightmost node of maximum depth

Example

1. Heap-Order: for every node m ≠root, key(m) ≥ key(parent(m))
2. Complete Binary Tree: left to right

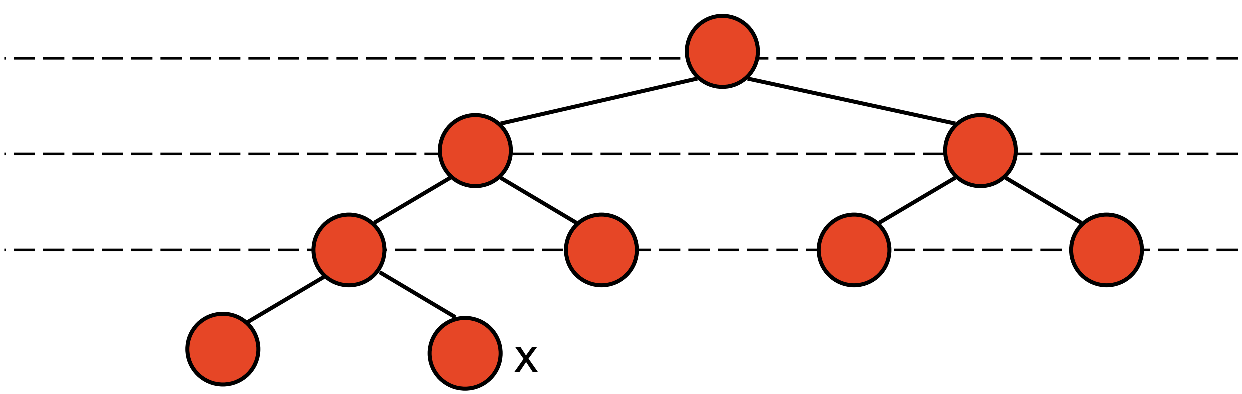


Minimum of a Heap

Fact: The root always holds the smallest key in the heap

Proof:

- Suppose the minimum key is at some internal node x
- Because of the heap property, as we move up the tree, the keys can only get smaller (assuming repeats, otherwise contradiction)
- If x is not the root, then its parent must also hold a smallest key
- Keep going until we reach the root

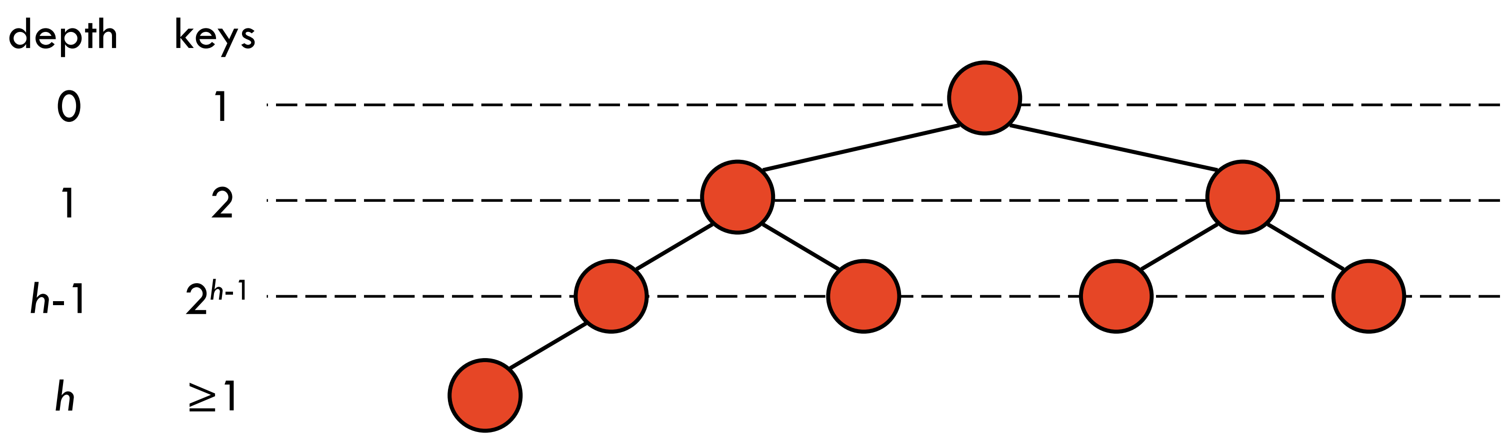


Height of a Heap

Fact: A heap storing n keys has height $\log n$

Proof:

- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \ldots, h - 1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1 = 2^h$
- Thus, $n \geq 2^h$, applying $\log_2$ on both sides, $\log_2 n \geq h$

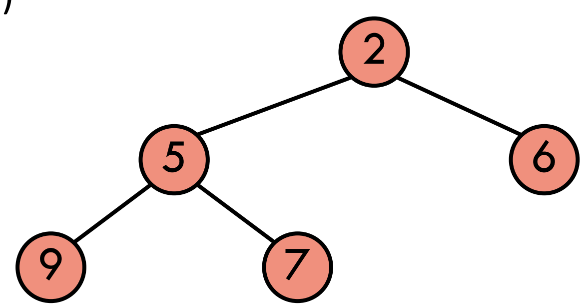


Insertion into a Heap

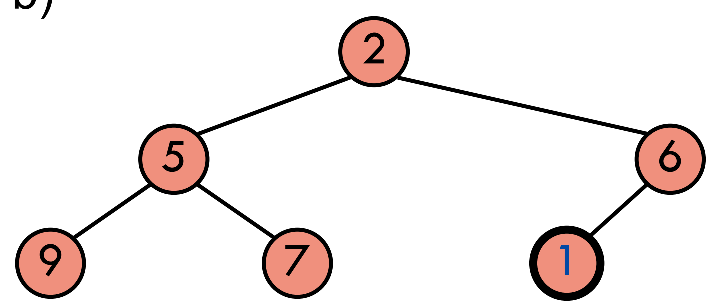
a) Create a new node with given key
b) Find location for new node (new last node)
c) Restore the heap-order property

insert(1)

a)



b)



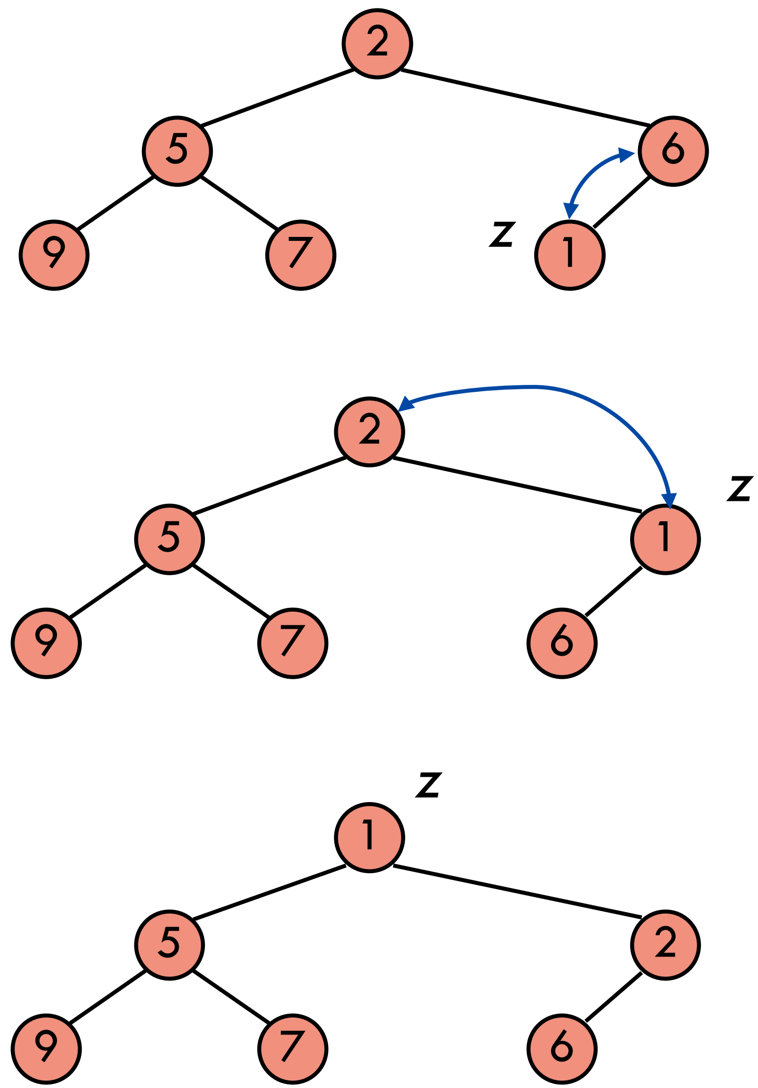
Upheap

Restore heap-order property by swapping keys along upward path from insertion point

```
def up_heap(z):
    while z ≠ root and key(parent(z)) > key(z) do
        swap key of z and parent(z)
        z ← parent(z)
```

Correctness: after swapping the subtree rooted at z has the heap-order property

Complexity: O(log n) time because the height of the heap is log n

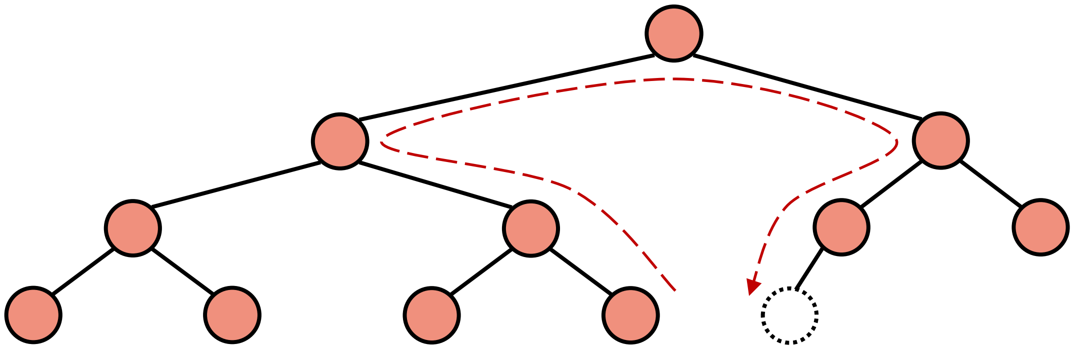


Finding the position for insertion

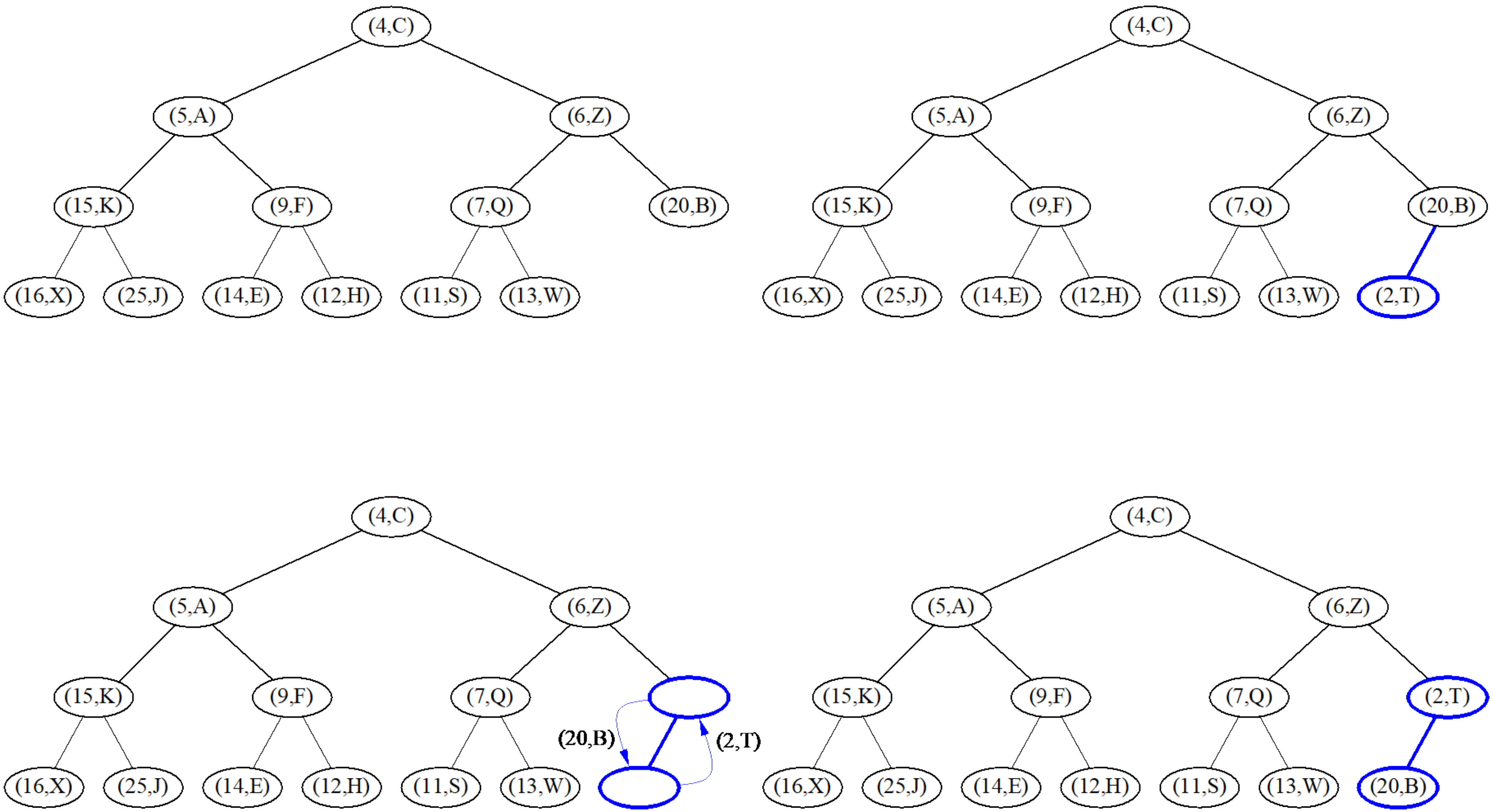
Two possible solutions:

1. Keep track of size of subtrees, or
2. Search starting from last node
   a) go up until a left child or the root is reached
   b) If we reach the root then need to open a new level
   c) otherwise, go to the sibling (right child of parent)
   d) go down left until a leaf is reached

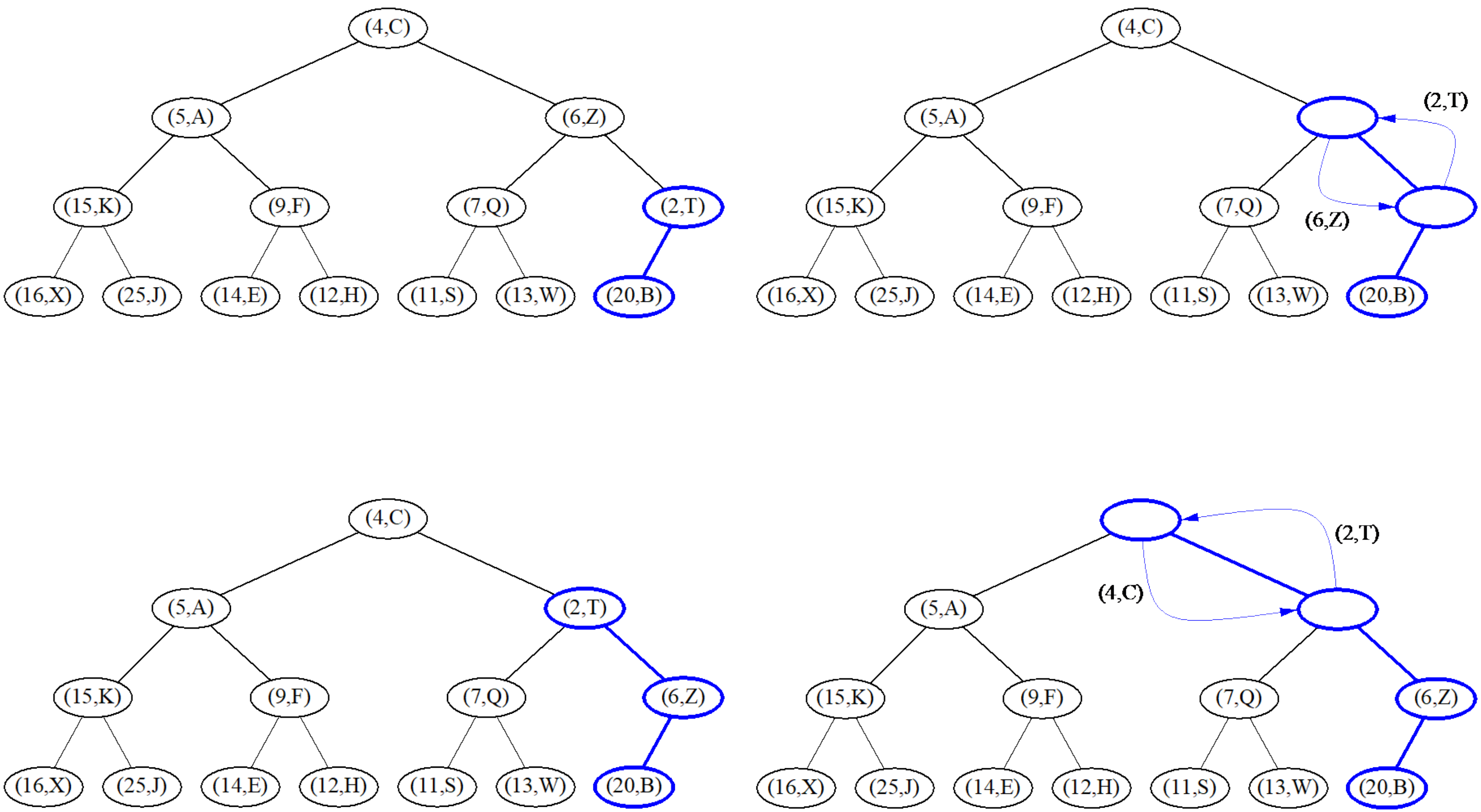
Complexity of this search is O(log n) because the height is log n. Thus, overall complexity of insertion is O(log n) time



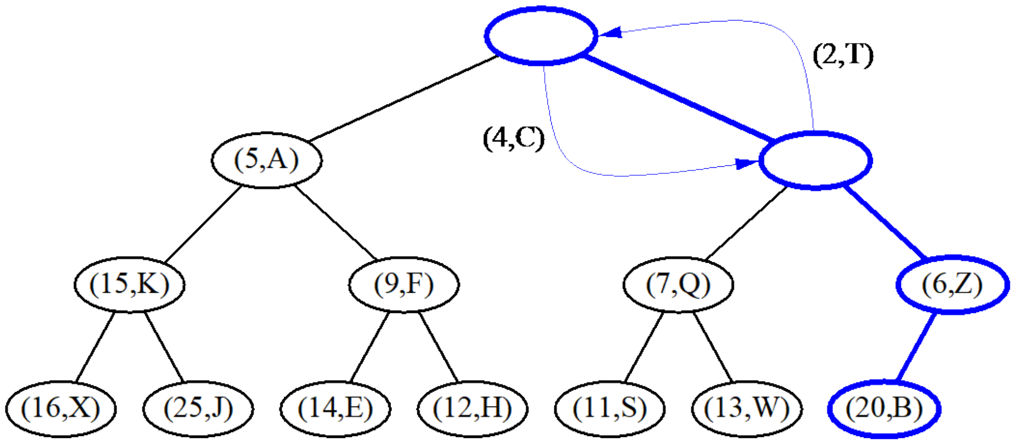
Example insertion (2,T)

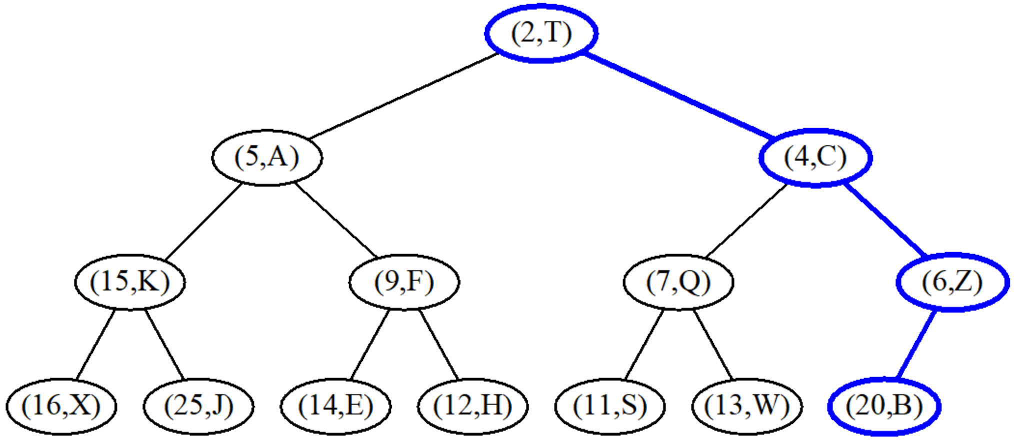


Example insertion (2,T) cont'd



Example insertion

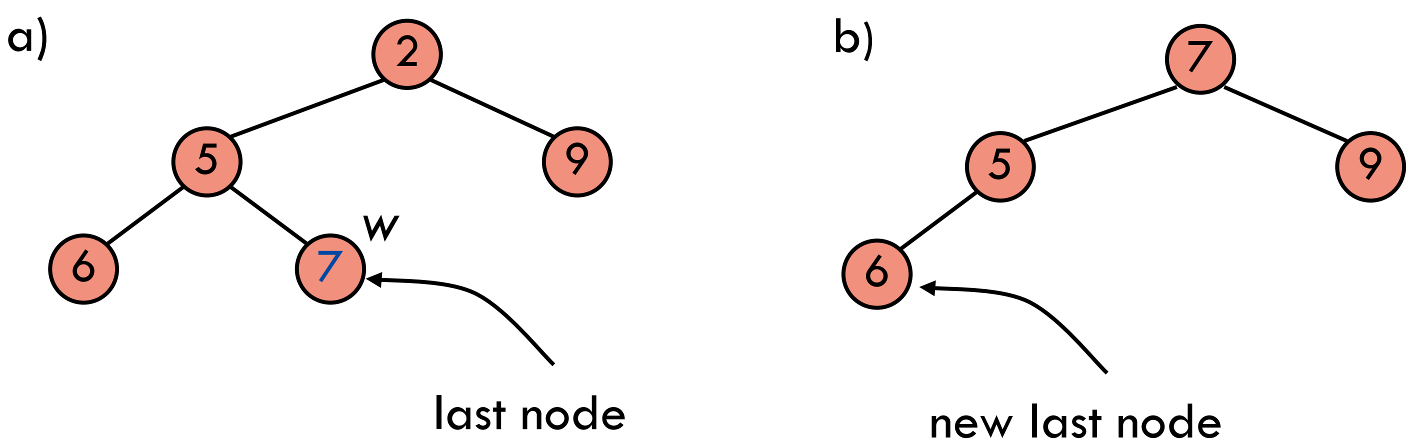




Removal from a Heap

a) Replace the root key with the key of the last node *w*
b) Delete *w*
c) Restore the heap-order property

remove_min()



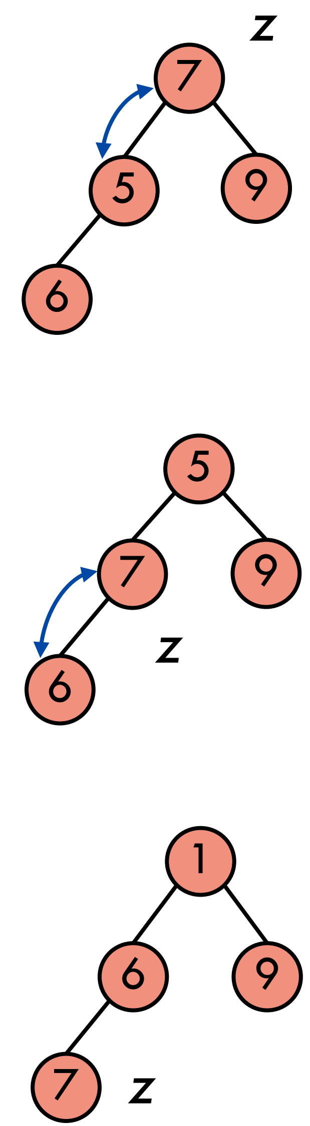
Downheap

Restore heap-order property by swapping keys along downward path from the root

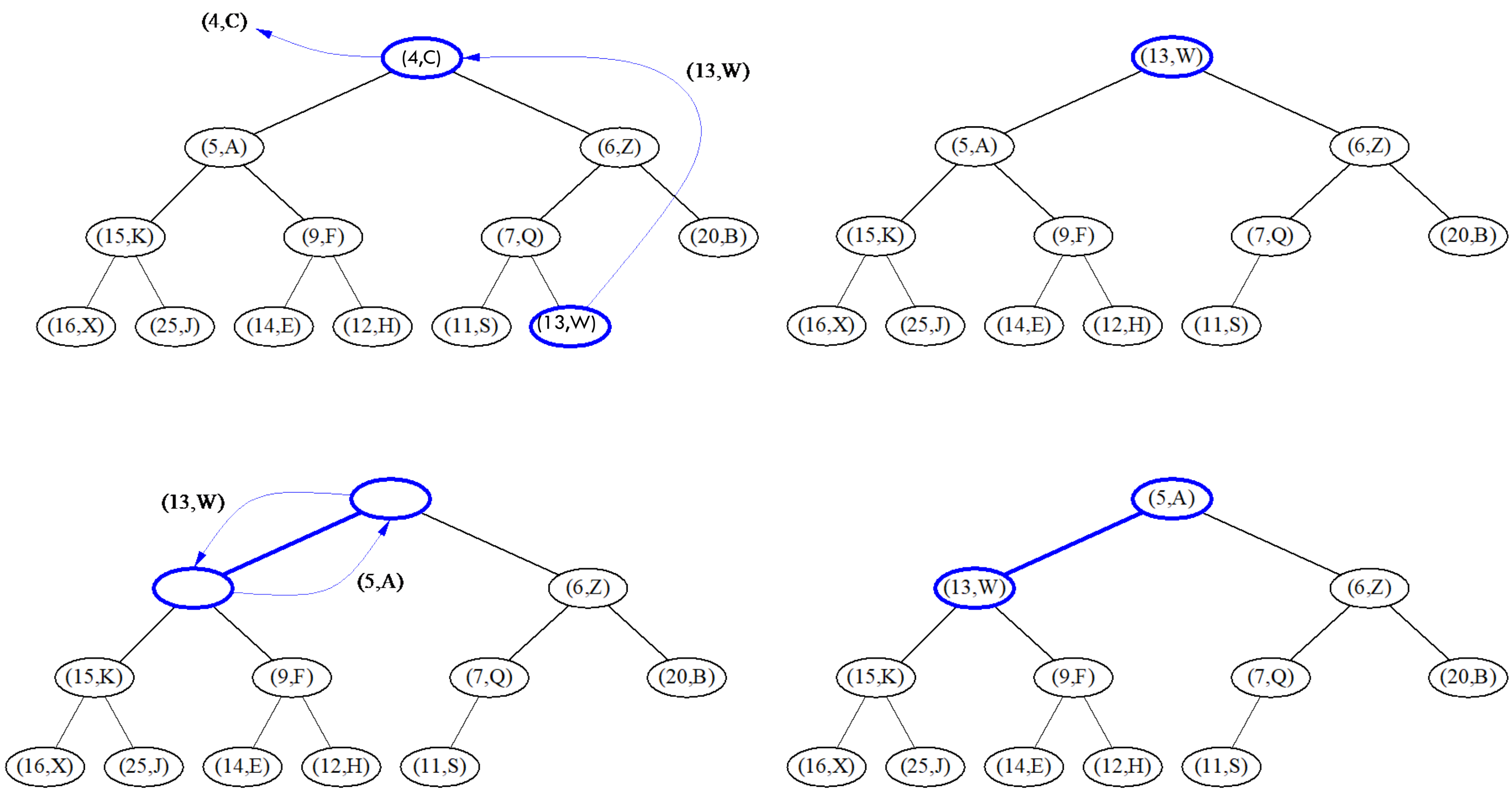
```
def down_heap(z):
    while z has child with
      key(child) < key(z) do
        x ← child of z with smallest key
        swap keys of x and z
        z ← x
```

Correctness: after swap z heap-order property is restored up to level of z

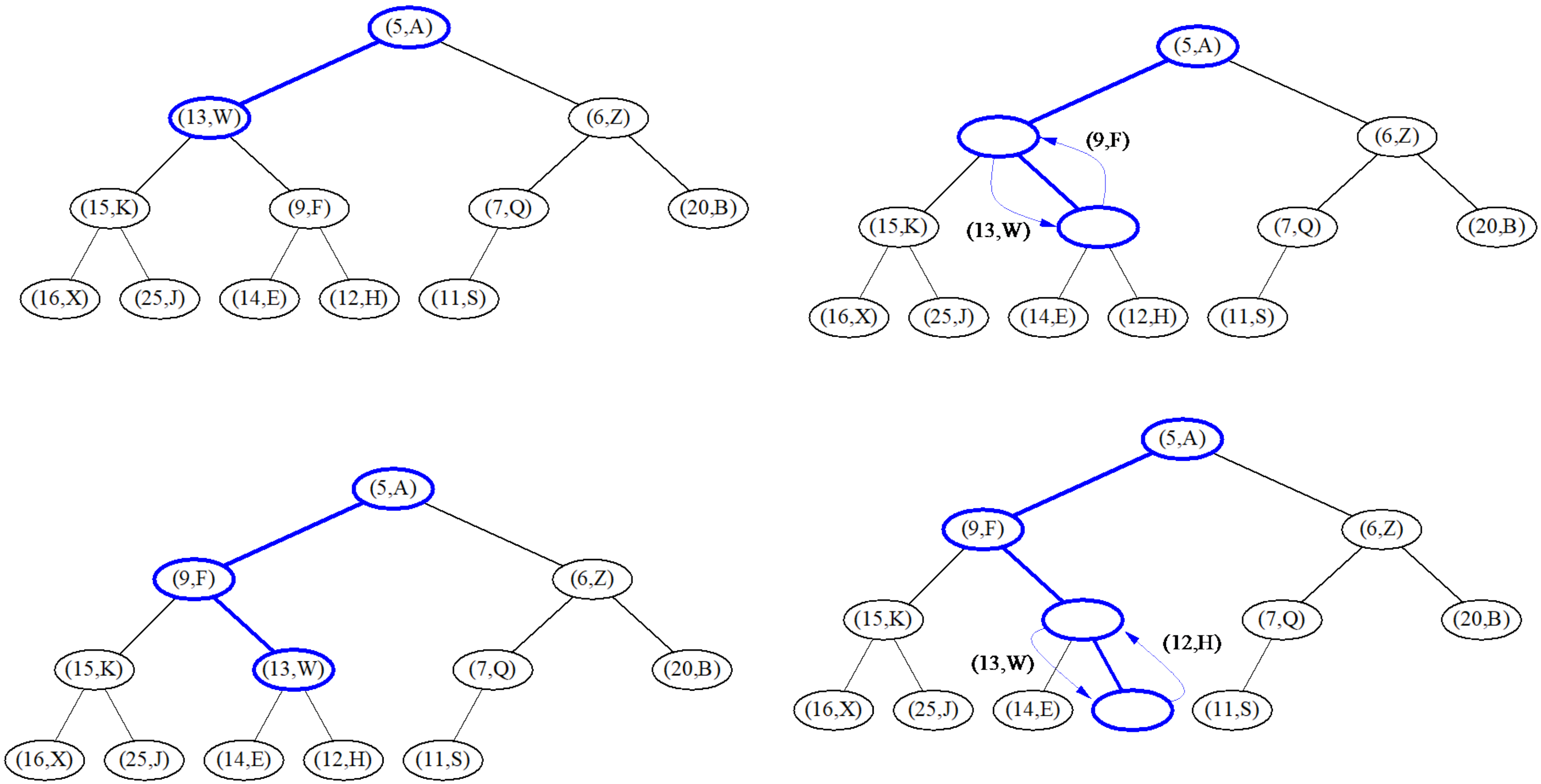
Complexity: O(log n) time because the height of the heap is log n



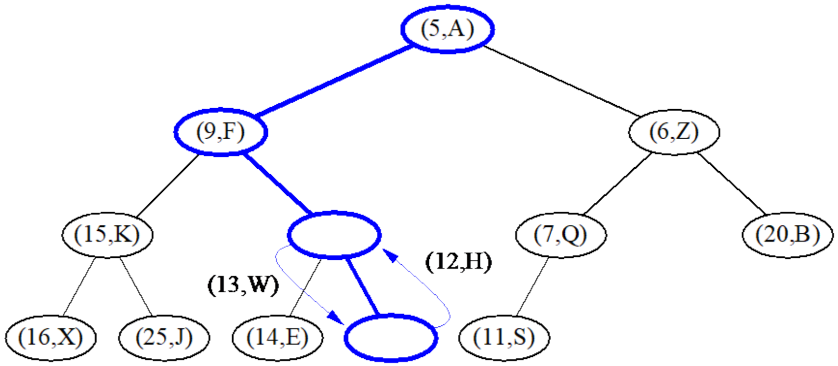
Example removal

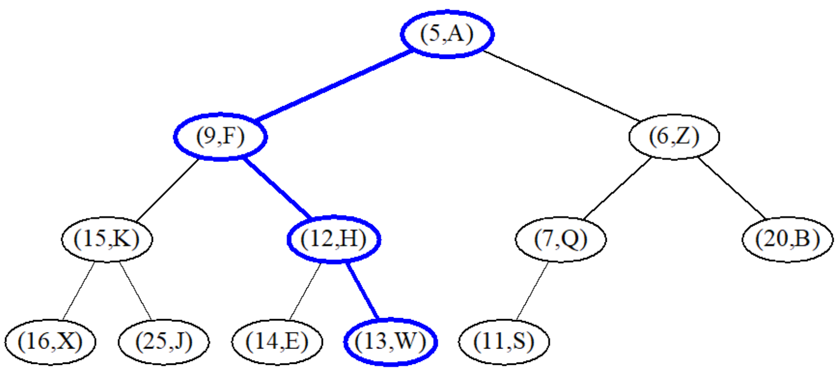


Example removal



Example removal



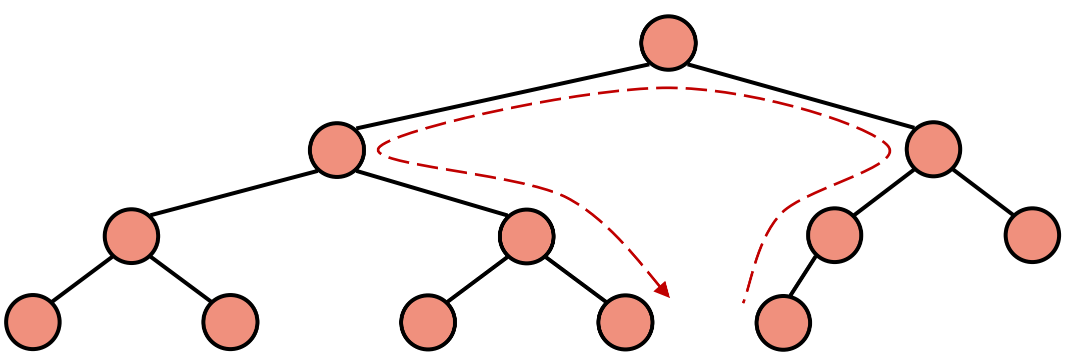


Finding next last node after deletion

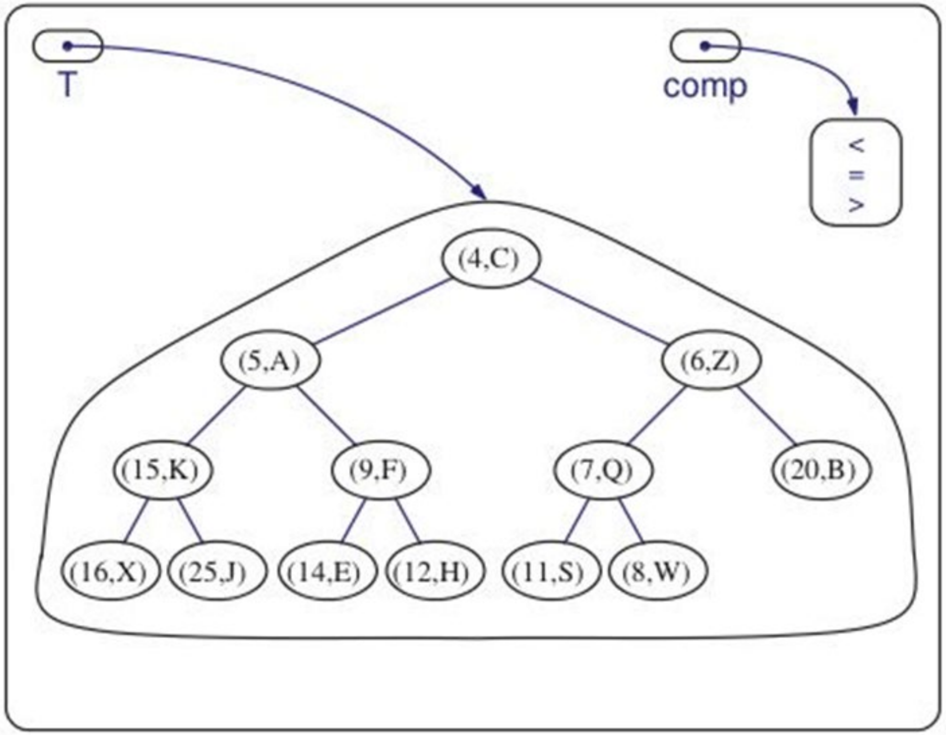
Two possible solutions:

1. Keep track of size of subtrees, or
2. start from the (old) last node
   a) go up until a right child or the root is reached
   b) if we reach the root then need to close a level
   c) otherwise, go to the sibling (left child of parent)
   d) go down right until a leaf is reached

Complexity of this search is O(log n) because the height is log n. Thus, overall complexity of deletion is O(log n) time



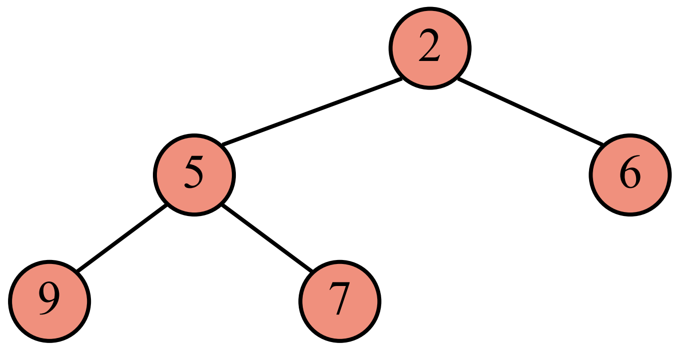
Heap-based implementation of a priority queue

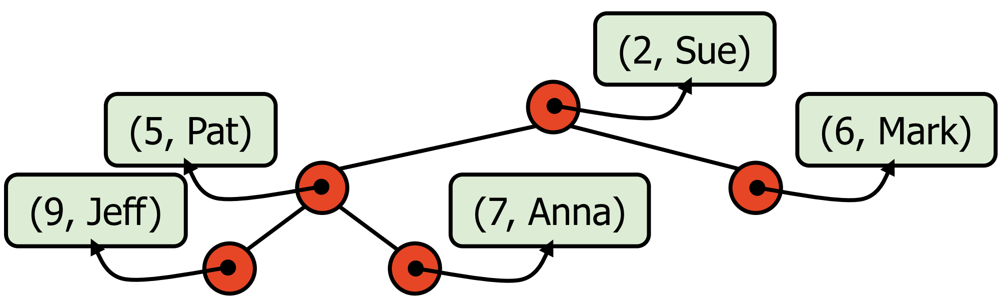


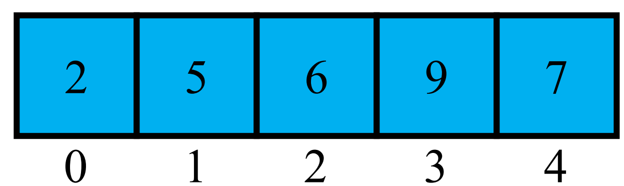
Operation	Time
size, isEmpty	$O(1)$
min,	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

Heap implementation

Heaps can be implemented as a linked structure or an array.

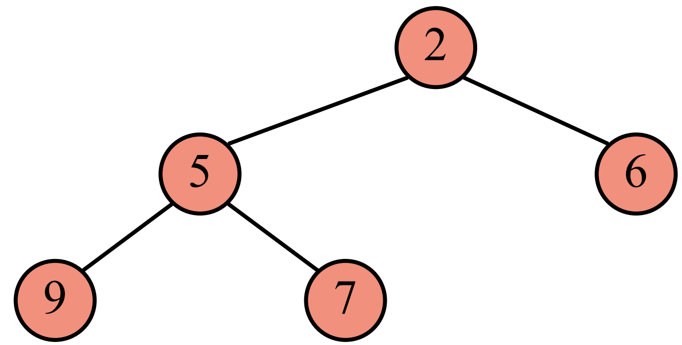






Heap-in-array implementation

We can represent a heap with *n* keys by means of an array of length *n*

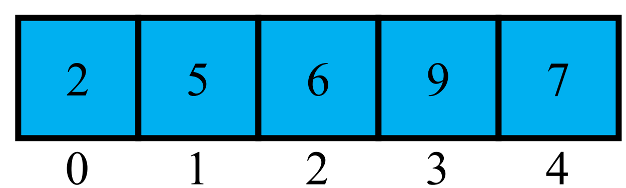


Special nodes:

– root is at 0
– last node is at n-1

For the node at index i:

– the left child is at index $2i+1$
– the right child is at index $2i+2$
– Parent is at index $\lfloor (i-1)/2 \rfloor$



Summary: Priority queue implementations

Method	Unsorted List	Sorted List	Heap	BST
size, isEmpty	$O(1)$	$O(1)$	$O(1)$	O(1)
insert	$O(1)$	$O(n)$	$O(\log n)$	O(log n)
min	$O(n)$	$O(1)$	$O(1)$	O(log n)
removeMin	$O(n)$	$O(1)$	$O(\log n)$	O(log n)

Building a heap in one go

Sometimes we have all the keys upfront. If we insert them one at a time, this can take $O(n \log n)$ time. However, there is a faster way to build the heap in this case.

```
def heapify (A):
    # turn A into a binary heap in place
    n ← size(A)
    for i from n-1 to 0 do
        down_heap(A, i)
```

If we let $h(i)$ be the height of the node corresponding to $A[i]$ then down_heap(A, i) takes $O(h(i))$ time.

Thus, the running time of the algorithm is $O(\sum_{i=0}^{n-1} h(i))$

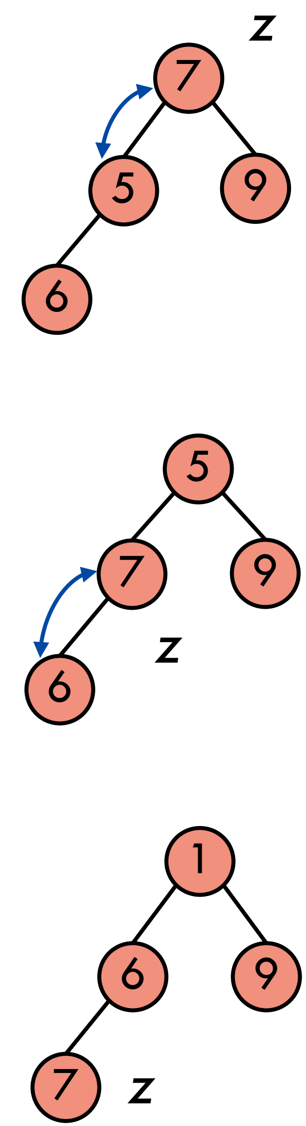
Downheap

Restore heap-order property by swapping keys along downward path from the root

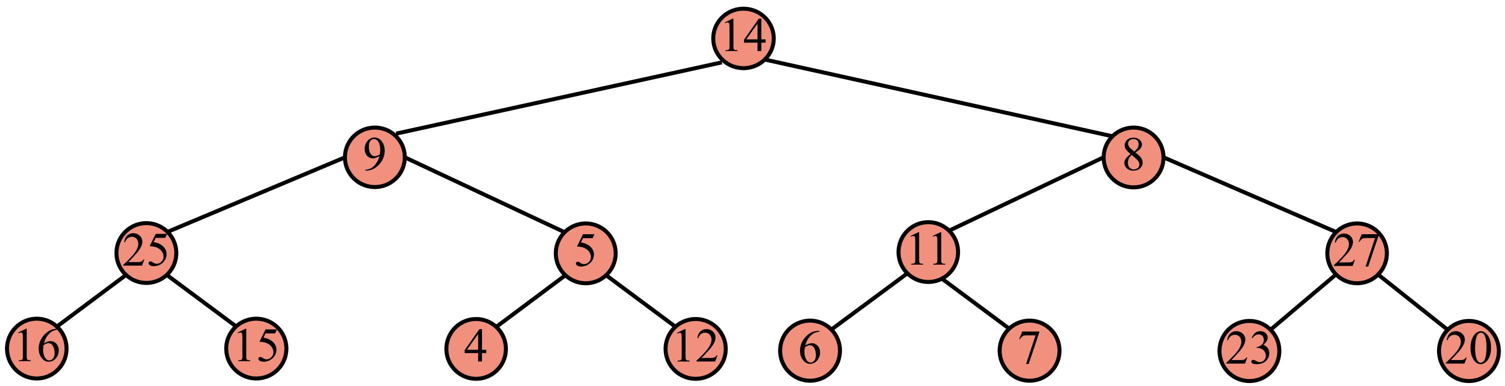
```
def down_heap(z):
    while z has child with
      key(child) < key(z) do
        x ← child of z with smallest key
        swap keys of x and z
        z ← x
```

Correctness: after swap z heap-order property is restored up to level of z

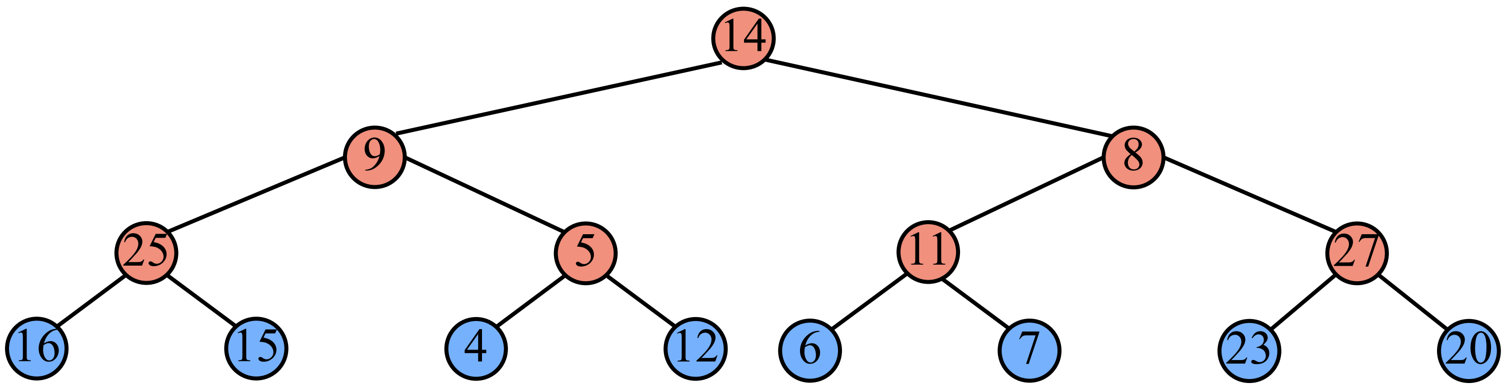
Complexity: O(log n) time because the height of the heap is log n



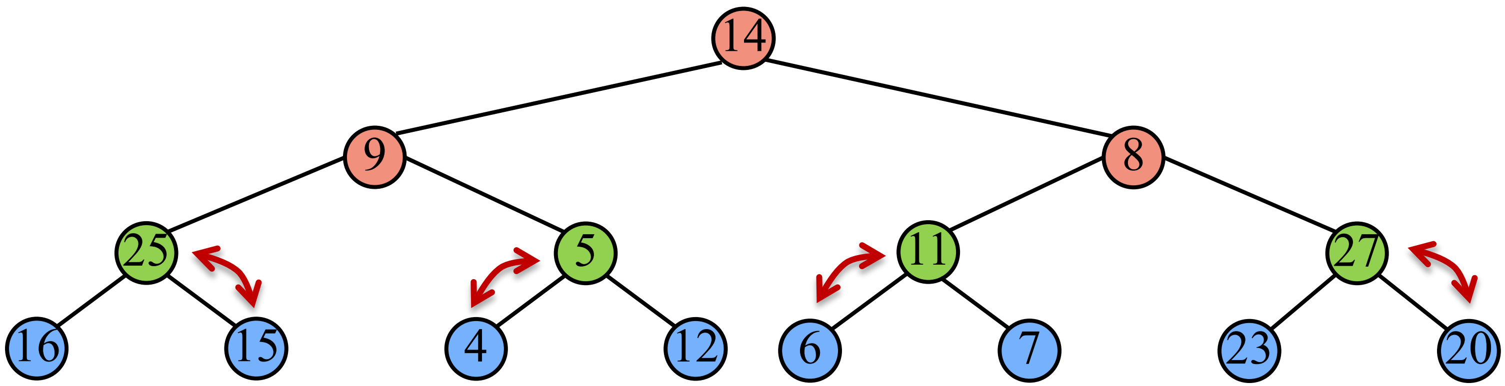
Building a heap in one go



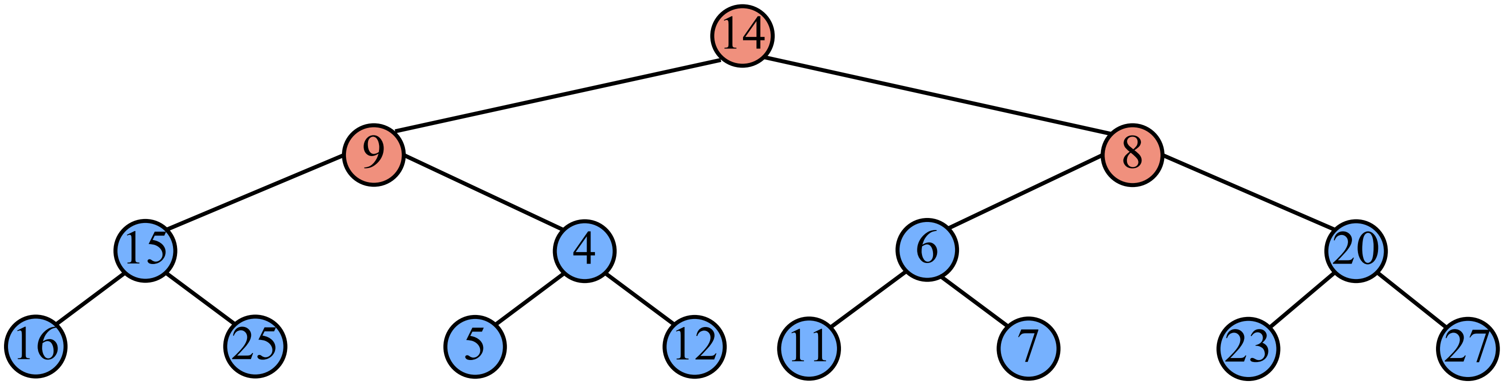
Building a heap in one go



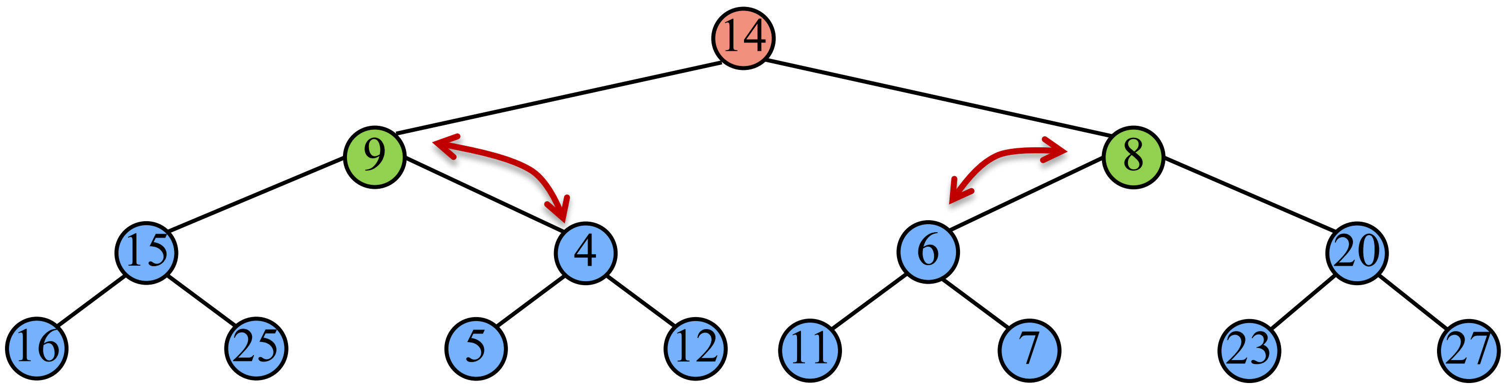
Building a heap in one go



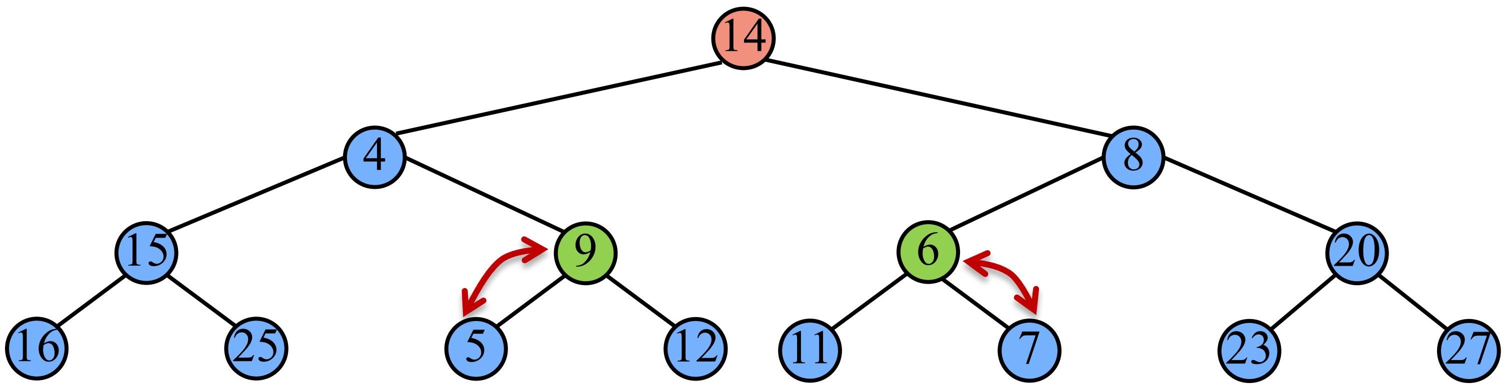
Building a heap in one go



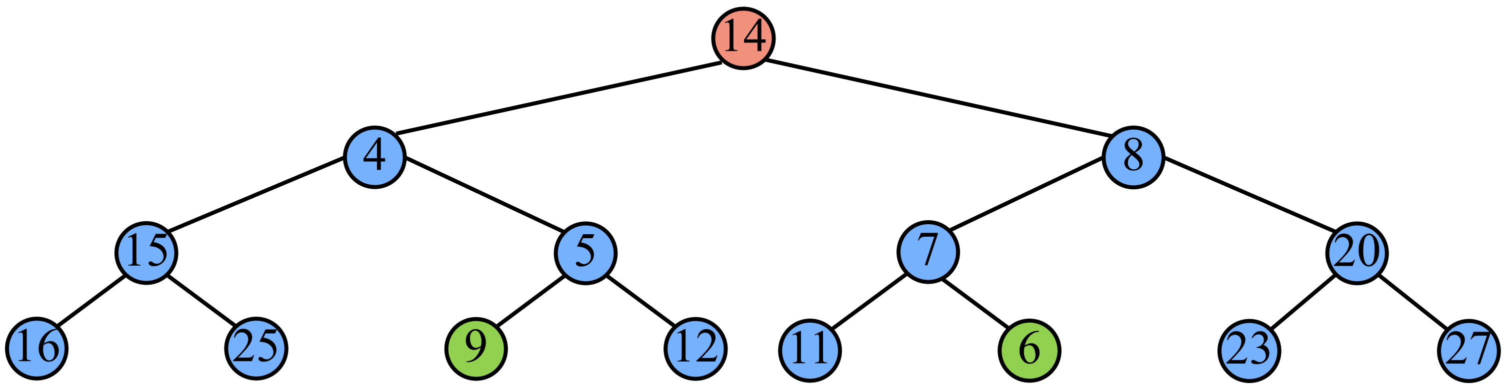
Building a heap in one go



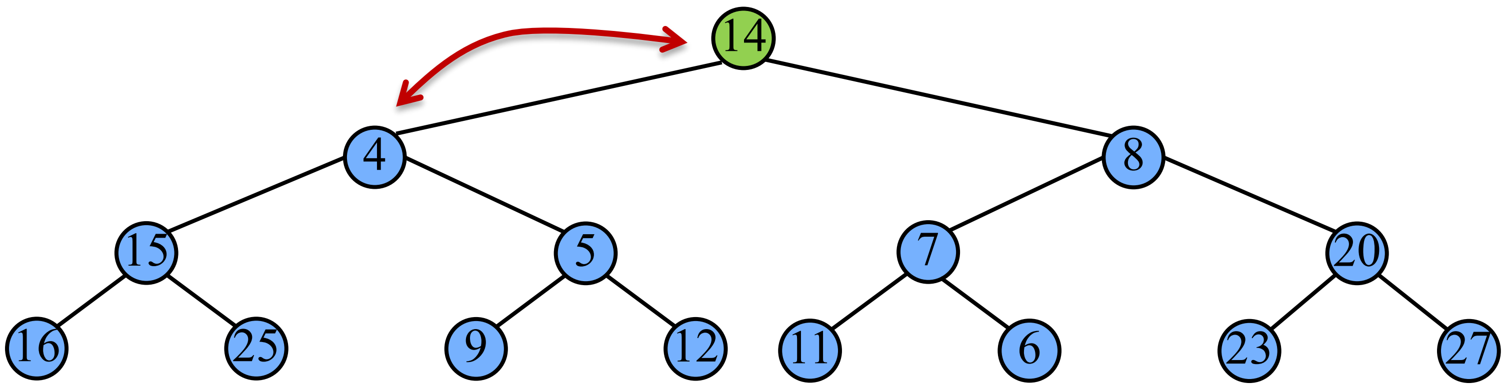
Building a heap in one go



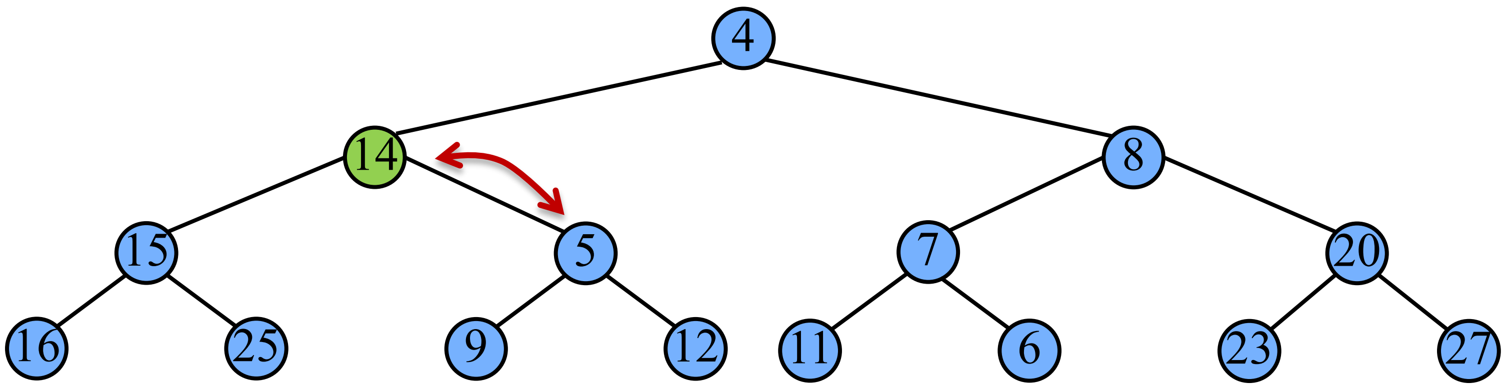
Building a heap in one go



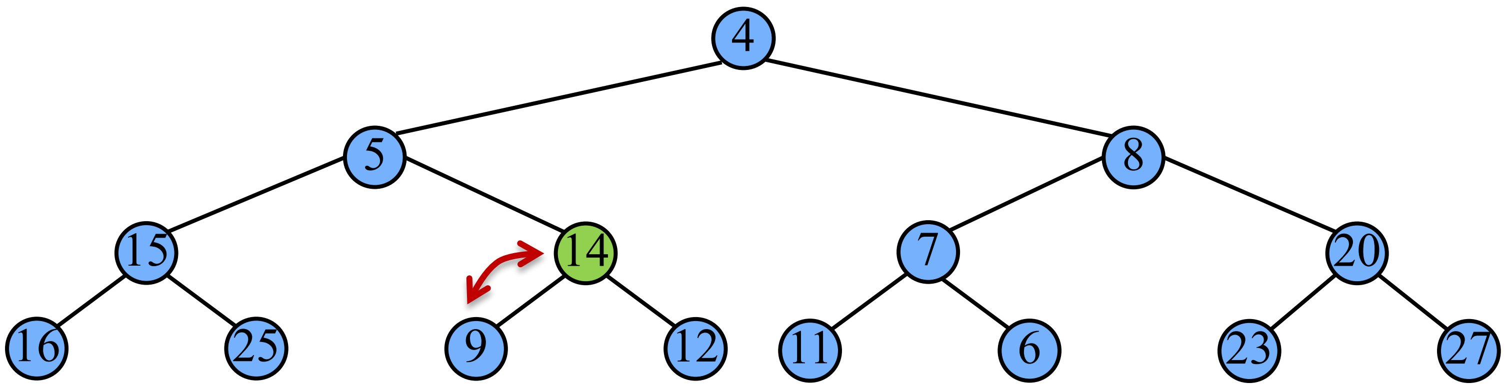
Building a heap in one go



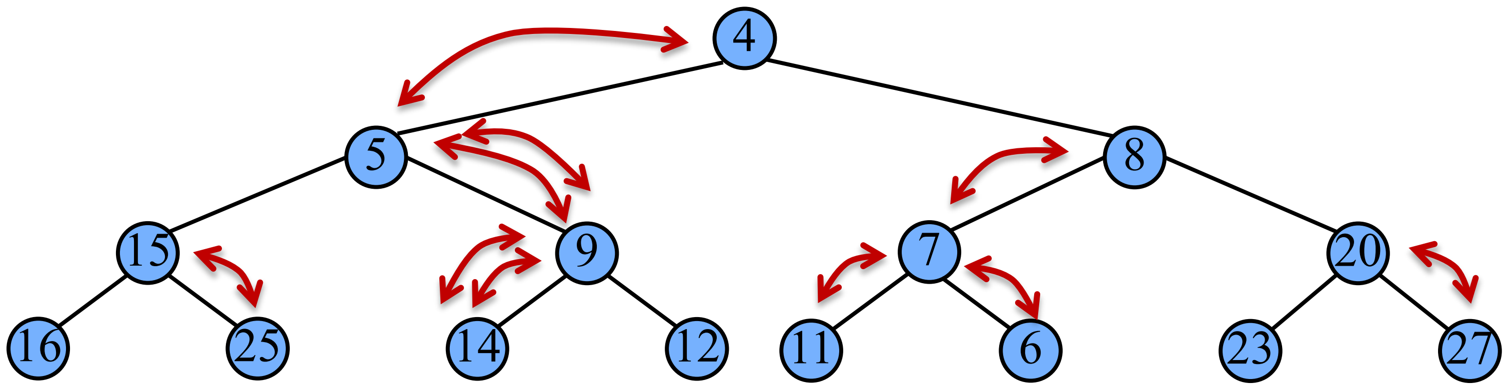
Building a heap in one go



Building a heap in one go



Building a heap in one go

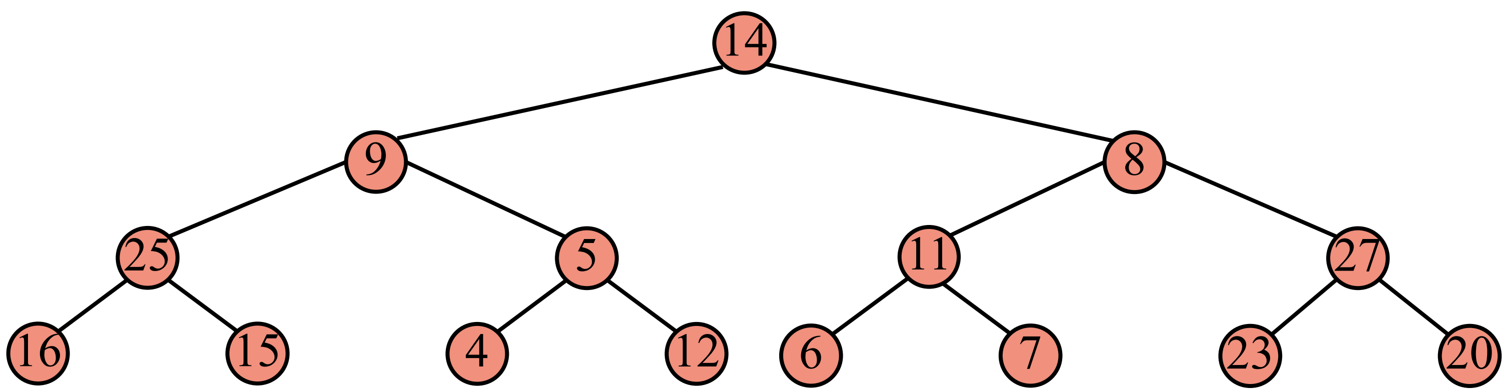


Building a heap in one go

Claim: The running time of the algorithm is $O(\sum_{i=0}^{n-1} h(i)) = O(n)$

For each node i in the tree construct a path of length h(i) by starting at node i, going **right once**, and then going left until we reach a leaf.

Claim: These paths are edge disjoint

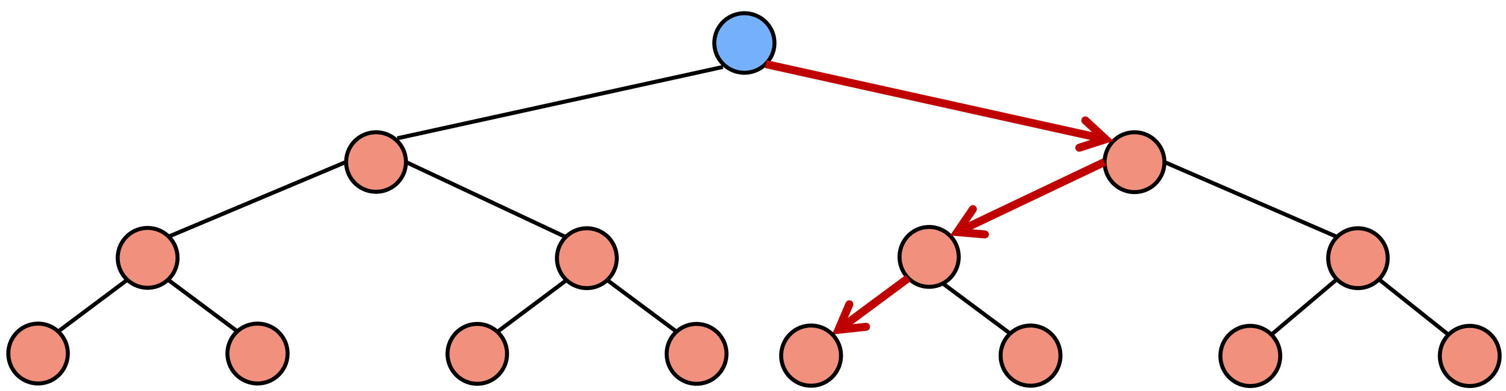


Building a heap in one go

Claim: The running time of the algorithm is $O(\sum_{i=0}^{n-1} h(i)) = O(n)$

For each node i in the tree construct a path of length h(i) by starting at node i, going **right once**, and then going left until we reach a leaf.

Claim: These paths are edge disjoint

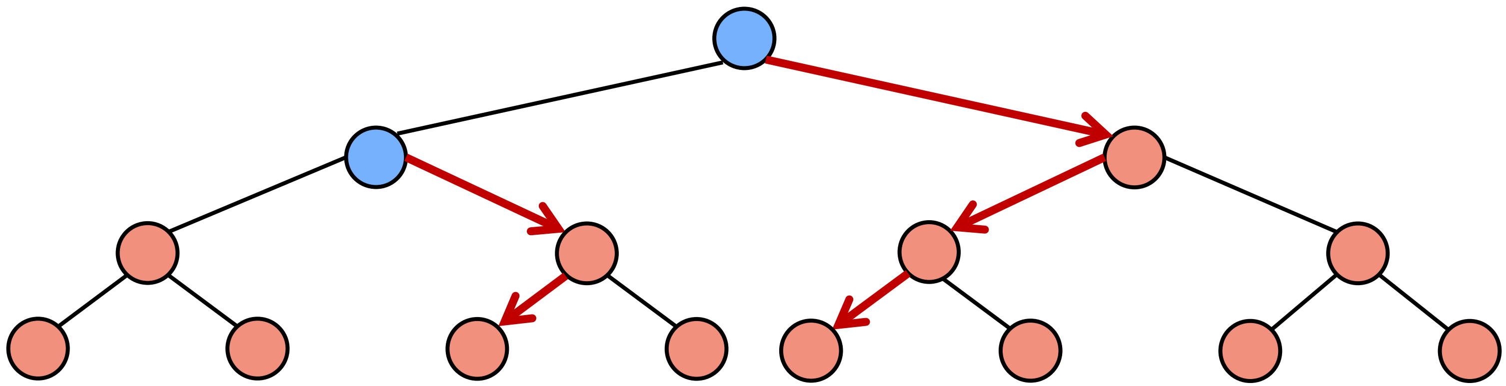


Building a heap in one go

Claim: The running time of the algorithm is $O(\sum_{i=0}^{n-1} h(i)) = O(n)$

For each node i in the tree construct a path of length h(i) by starting at node i, going **right once**, and then going left until we reach a leaf.

Claim: These paths are edge disjoint

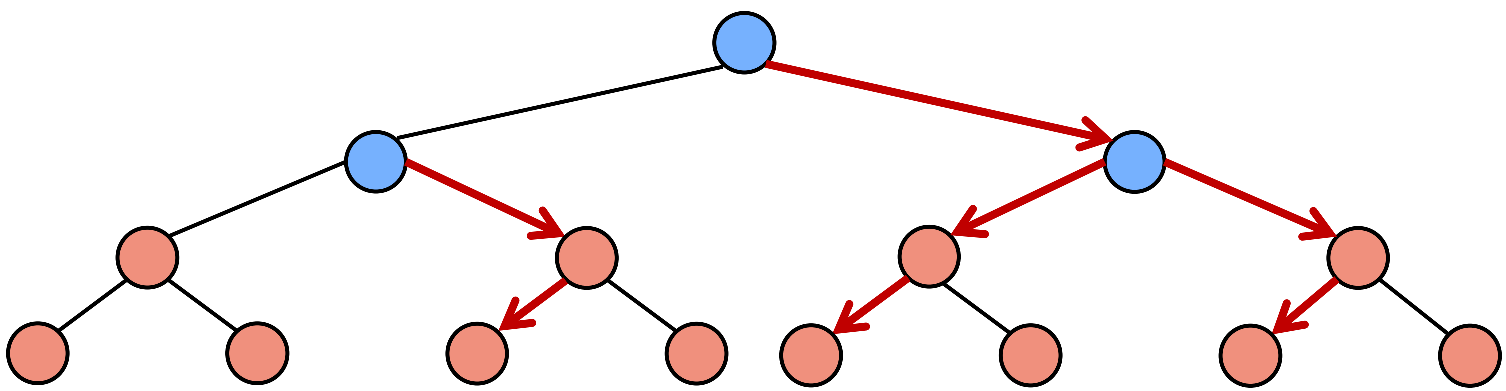


Building a heap in one go

Claim: The running time of the algorithm is $O(\sum_{i=0}^{n-1} h(i)) = O(n)$

For each node i in the tree construct a path of length h(i) by starting at node i, going **right once**, and then going left until we reach a leaf.

Claim: These paths are edge disjoint

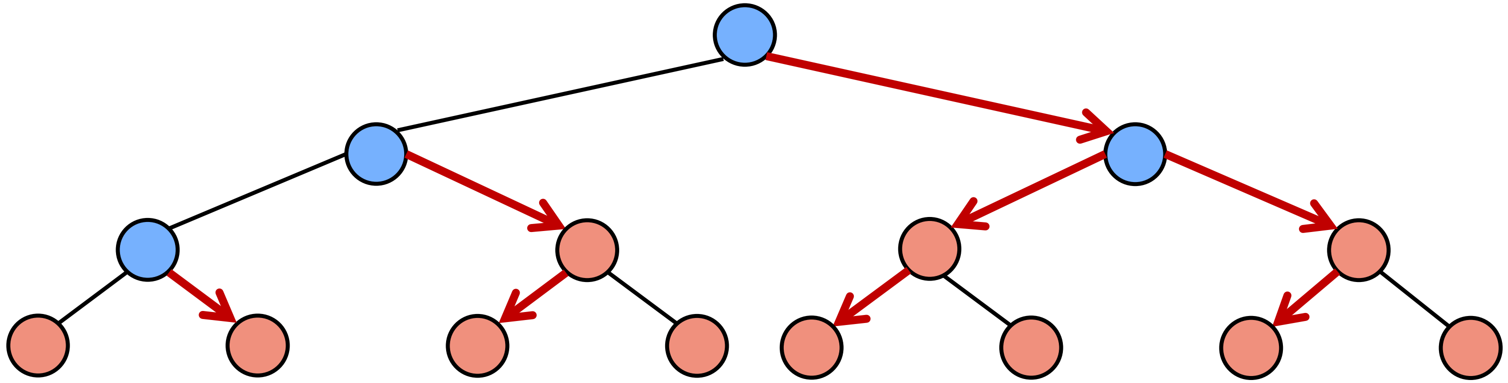


Building a heap in one go

Claim: The running time of the algorithm is $O(\sum_{i=0}^{n-1} h(i)) = O(n)$

For each node i in the tree construct a path of length $h(i)$ by starting at node i, going **right once**, and then going left until we reach a leaf.

Claim: These paths are edge disjoint

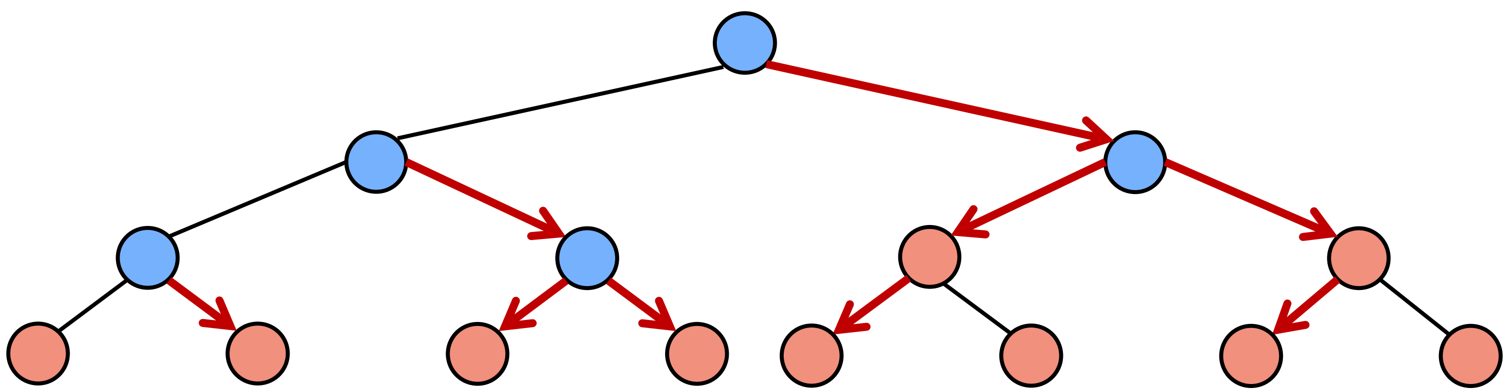


Building a heap in one go

Claim: The running time of the algorithm is $O(\sum_{i=0}^{n-1} h(i)) = O(n)$

For each node i in the tree construct a path of length h(i) by starting at node i, going **right once**, and then going left until we reach a leaf.

Claim: These paths are edge disjoint

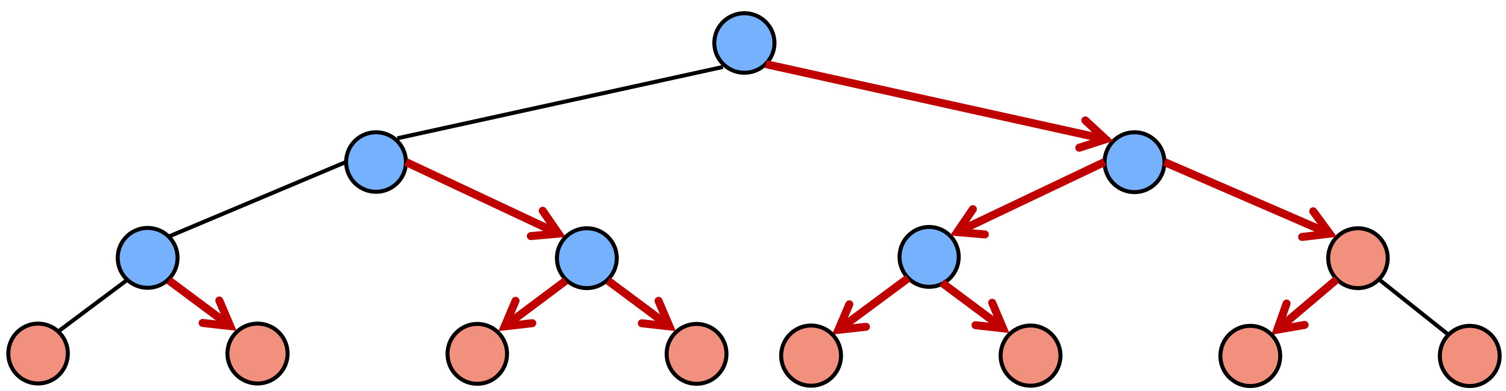


Building a heap in one go

Claim: The running time of the algorithm is $O(\sum_{i=0}^{n-1} h(i)) = O(n)$

For each node i in the tree construct a path of length h(i) by starting at node i, going **right once**, and then going left until we reach a leaf.

Claim: These paths are edge disjoint

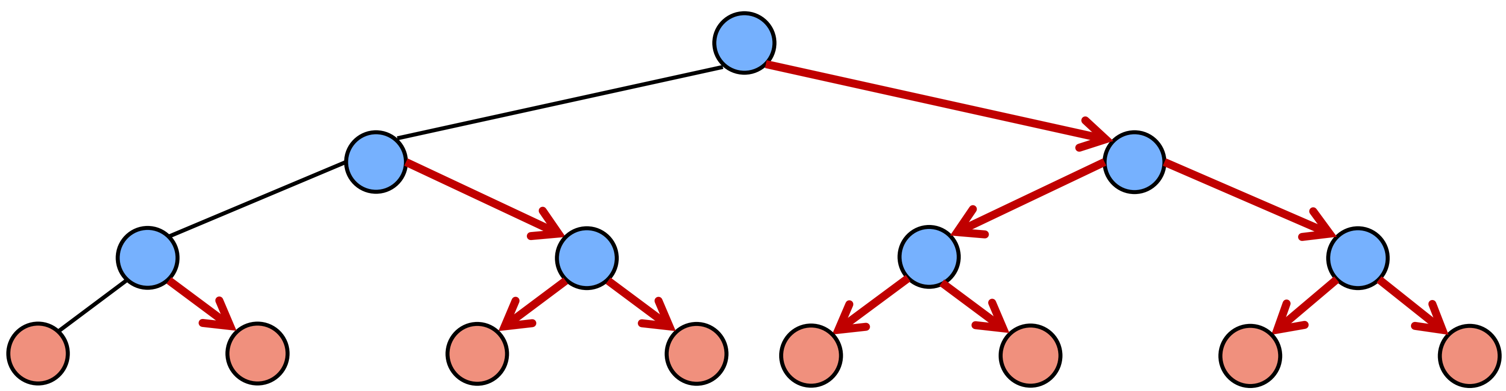


Building a heap in one go

Claim: The running time of the algorithm is $O(\sum_{i=0}^{n-1} h(i)) = O(n)$

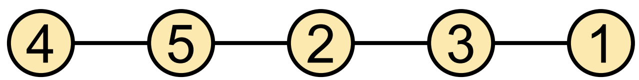
For each node i in the tree construct a path of length h(i) by starting at node i, going **right once**, and then going left until we reach a leaf.

Claim: These paths are edge disjoint



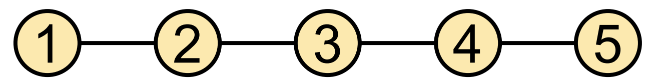
Sequence-based Priority Queue

Unsorted list implementation



- insert in O(1) time since we can insert the item at the beginning or end of the sequence
- remove_min and min in O(n) time since we have to traverse the entire list to find the smallest key

Sorted list implementation



- insert in O(n) time since we have to find the place where to insert the item
- remove_min and min in O(1) time since the smallest key is at the beginning

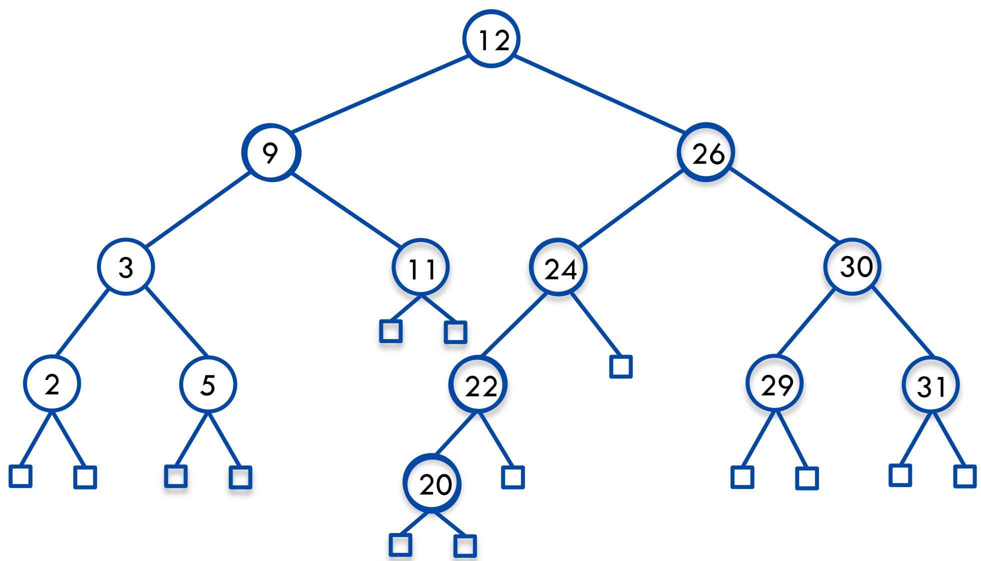
Method	Unsorted List	Sorted List
size, isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

BST-based Priority Queue

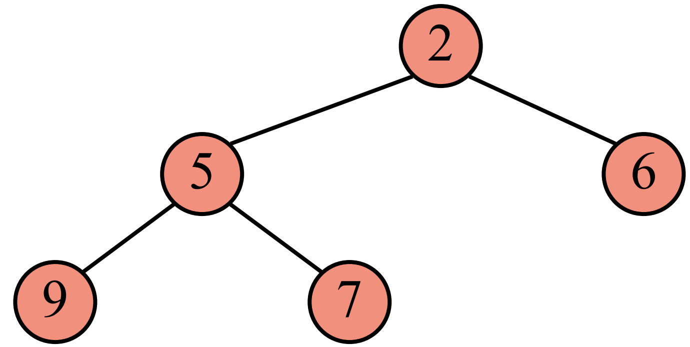
– insert in O(log n) time

– remove_min and min in O(log n) time

Method	Unsorted List	Sorted List	BST
size, isEmpty	$O(1)$	$O(1)$	O(1)
insert	$O(1)$	$O(n)$	O(log n)
min, removeMin	$O(n)$	$O(1)$	O(log n)



Heap-based Priority Queue



insert in O(log n) time

remove_min and min in O(log n) time

Can be built in O(n) time

Summary:

Method	Unsorted List	Sorted List	Heap	BST
size, isEmpty	$O(1)$	$O(1)$	$O(1)$	O(1)
insert	$O(1)$	$O(n)$	$O(\log n)$	O(log n)
min	$O(n)$	$O(1)$	$O(1)$	O(log n)
removeMin	$O(n)$	$O(1)$	$O(\log n)$	O(log n)

Priority Queue Sorting

We can use a priority queue to sort a list of keys:

1. iteratively insert keys into an empty priority queue
2. iteratively remove_min to get the keys in sorted order

Complexity analysis:

- n insert operations
- n remove_min operations

```
def priority_queue_sorting(A):
    pq ← new priority queue
    n ← size(A)
    for i in [0:n] do
        pq.insert(A[i])
    for i in [0:n] do
        A[i] ← pq.remove_min()
```

Sorting using a:

List $O(n^2)$

BST $O(n \log n)$

Heaps $O(n \log n)$

Method	Unsorted List	Sorted List	Heap	BST
size, isEmpty	$O(1)$	$O(1)$	$O(1)$	O(1)
insert	$O(1)$	$O(n)$	$O(\log n)$	O(log n)
min	$O(n)$	$O(1)$	$O(1)$	O(log n)
removeMin	$O(n)$	$O(1)$	$O(\log n)$	O(log n)

Selection-Sort

Variant of pq-sort using unsorted sequence implementation:

1. inserting elements with n insert operations takes $O(n)$ time
2. removing elements with n remove_min operations takes $O(n^2)$

Can be done in place
(no need for extra space)

Top level loop invariant:

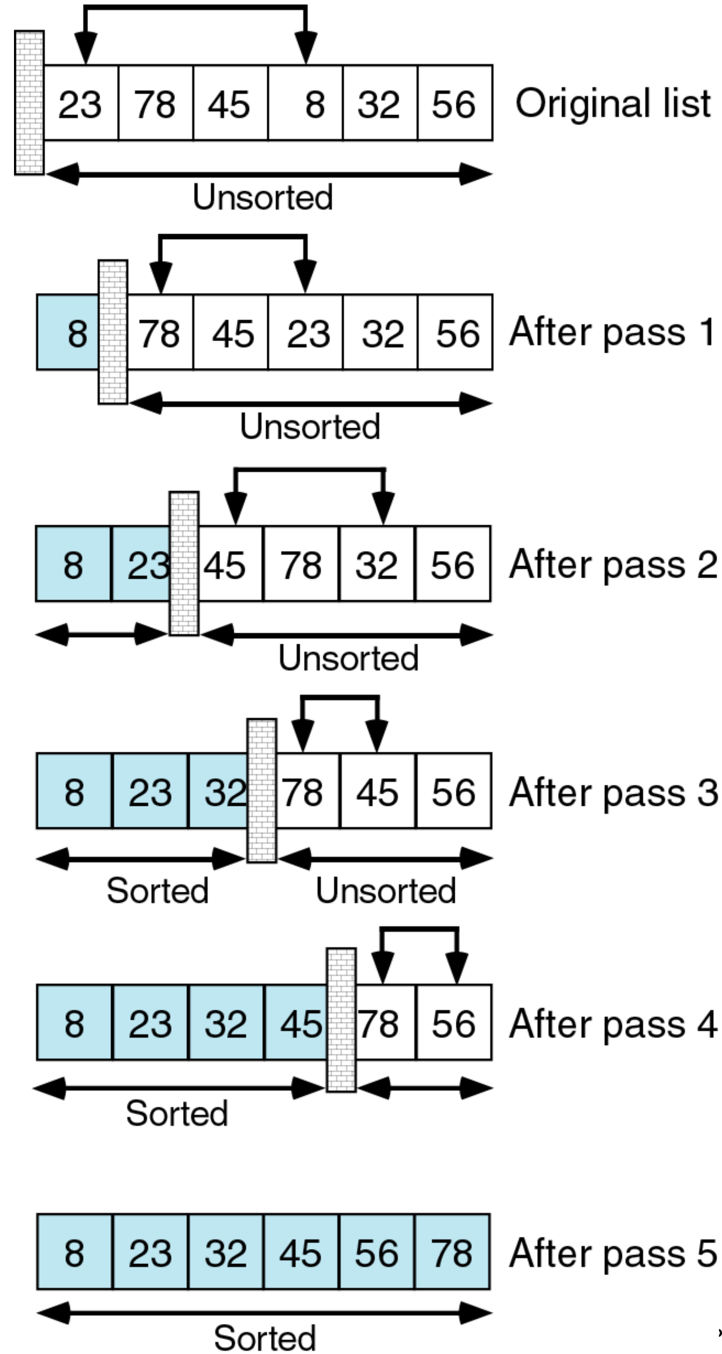
- A[0:i] is sorted
- A[i:n] is the priority queue and all ⩾ A[i-1]

```
def selection_sort(A):
    n ← size(A)
    for i in [0:n] do
        # find s ⩾ i minimizing A[s]
        s ← i
        for j in [i+1:n] do
            if A[j] < A[s] then
                s ← j
        # swap A[i] and A[s]
        A[i], A[s] ← A[s], A[i]
```

Selection-Sort Example

```
def selection_sort(A):
    n ← size(A)
    for i in [0:n] do
        s ← i
        for j in [i+1:n] do
            if A[j] < A[s] then
                s ← j
        A[i], A[s] ← A[s], A[i]
```

Running time: $O(n^2)$



Insertion-Sort

Variant of pq-sort using sorted sequence implementation:

1. inserting elements with n insert operations takes $O(n^2)$ time
2. removing elements with n remove_min operations takes $O(n)$

Can be done in place
(no need for extra space)

Top level loop invariant:

– A[0:i] is the priority queue (and thus sorted)
– A[i:n] is yet-to-be-inserted

```
def insertion_sort(A):
    n ← size(A)
    for i in [1:n] do
        x ← A[i]
        # move forward entries > x
        j ← i
        while j > 0 and x < A[j-1] do
            A[j] ← A[j-1]
            j ← j – 1
        # if j>0 ⇒ x ≥ A[j-1]
        # if j<i ⇒ x < A[j+1]
        A[j] ← x
```

Insertion-Sort Example

```
def insertion_sort(A):
    n ← size(A)
    for i in [1:n] do
        x ← A[i]
        j ← i
        while j > 0 and x < A[j-1] do
            A[j] ← A[j-1]
            j ← j – 1
        A[j] ← x
```

4	2	3	5	1	9	6	6
2	4	3	5	1	9	6	6
2	3	4	5	1	9	7	6
2	3	4	5	1	9	7	6
2	3	4	1	5	9	7	6
2	3	1	4	5	9	7	6
2	1	3	4	5	9	7	6
1	2	3	4	5	9	7	6
1	2	3	4	5	9	7	6
1	2	3	4	5	7	9	6
1	2	3	4	5	7	6	9
1	2	3	4	5	6	7	9

Running time: $O(n^2)$

- What if the input is sorted?
- What if the input is sorted in descending order?

Heap-Sort

Consider a priority queue with n items implemented with a heap:

- the space used is $O(n)$
- methods insert and remove_min take O(log n)

Recall that priority-queue sorting uses:

- n insert ops
- n remove_min ops

The n insertions can be done in one go in O(n) time.

Heap-sort is the version of priority-queue sorting that implements the priority queue with a heap. It runs in O(n log n) time.

Refinements and Generalization

Heap-sort can be arranged to work in place using part of the array for the output and part for the priority queue

A heap on n keys can be constructed in O(n) time. But the n remove_min still take O(n log n) time

Sometimes it is useful to support a few more operations (all given a pointer to e):

– remove(e): Remove item e from the priority queue
– replace_key(e, k): update key of item e with k
– replace_value(e, v): update value of item e with v

Method	Unsorted List	Sorted List	Heap	BST
size, isEmpty	$O(1)$	$O(1)$	$O(1)$	O(1)
insert	$O(1)$	$O(n)$	$O(\log n)$	O(log n)
min	$O(n)$	$O(1)$	$O(1)$	O(log n)
removeMin	$O(n)$	$O(1)$	$O(\log n)$	O(log n)
remove	$O(1)$	$O(1)$	$O(\log n)$	O(log n)
replaceKey	$O(1)$	$O(n)$	$O(\log n)$	O(log n)
replaceValue	$O(1)$	$O(1)$	$O(1)$	O(1)

Sorting: running times in practice (ms)

Array Size	Selection Sort	Insertion Sort	Heap Sort
1000	5,64	3,90	1,33
10000	80,45	25,97	3,85
50000	1661,01	348,15	12,44
100000	6792,87	1379,62	19,62
500000	164748,01	34541,45	87,24
1000000	670833,80	138192,96	248,09

~11 mins

~0.25 sec