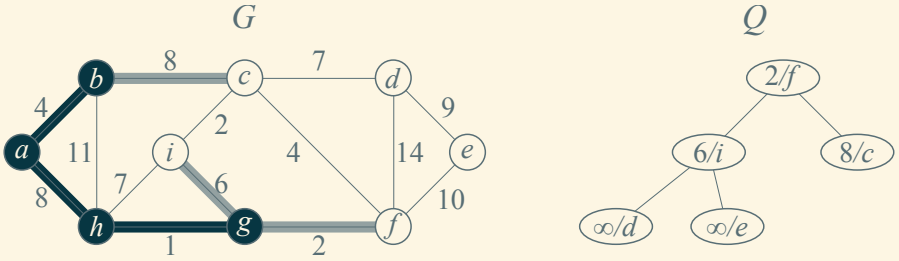
Lesson 04: Heaps and Prim's Minimum-Spanning-Tree Algorithm

Michael T. Gastner (7 March 2023)



Disclaimer: These slides are based on and occasionally quote from 'Introduction to Algorithms' (3rd ed.) by Cormen et al. (2009), MIT Press.

Review of Previous Lesson

Disjoint-Set Forests and Kruskal's Minimum-Spanning-Tree Algorithm

Last time, we studied Kruskal's algorithm, which grows a minimum spanning tree by adding edges in ascending order of weight. Whenever a candidate edge would close a cycle, the edge is skipped, and we proceed with checking the next-larger edge. This operation results in a dynamic forest in which the number of trees steadily decreases from $|V|$ to 1.

For Kruskal's algorithm to be fast, it needs to update the connected components of the forest (i.e. the trees) quickly while edges are added. We learnt that a disjoint-set forest, which represents sets in a collection as trees, is a suitable data structure for this task.

2 / 34

Learning Objectives

By the end of this lesson, you should be able to ...

- Recall the min-heap and max-heap properties.
- Explain the heap procedures needed for the heapsort algorithm and for maintaining priority queues.
- Apply heap procedures and Prim's algorithm to sample data.
- State how quickly the worst-case running times of heap procedures grow as a function of the size of the input.
- Derive the worst-case running time of Dijkstra's algorithm.

3 / 34

Heaps

Data Structure for Dynamic Sorting

A **(binary) heap** is an array object of keys that we can view as a nearly complete binary tree whose keys can be sorted (e.g. numerically or alphabetically). Nodes in the tree must satisfy either the **max-heap property** or the **min-heap property**:

- The max-heap property is that the key of every non-root node is not greater than the key of its parent. Consequently, the root has the maximal key in the heap.
- The min-heap property is that the key of every non-root node is not less than the key of its parent. Consequently, the root has the minimal key in the heap.

Heaps are useful for keeping a dynamically changing set of values in sorted order.

4 / 34

Representing a heap using an array

Array Attributes

An array A that represents a heap must have two attributes:

- $A.length$: number of elements in A
- $A.heap\text{-}size$: number of elements in the heap that are stored in A

That is, although $A[1, \ldots, A.length]$ may contain values, only the elements in $A[1, \ldots, A.heap\text{-}size]$, where $0 \leq A.heap\text{-}size \leq A.length$, are valid elements of the heap.

The root of the tree is $A[1]$.

5 / 34

Representing a heap using an array

Accessing Parents and Children in the Tree

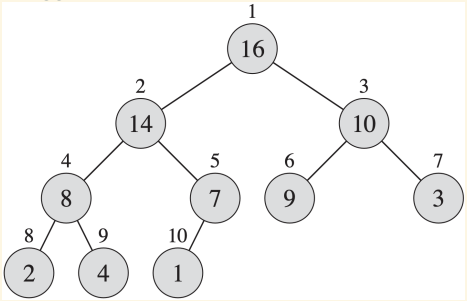
Here is pseudocode for the procedures for accessing the parent, the left child and the right child of a node i.

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

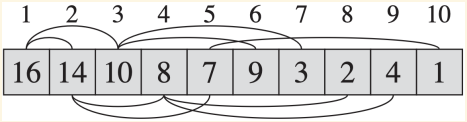
LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

Tree:



Array:



6 / 34

Representing a heap using an array

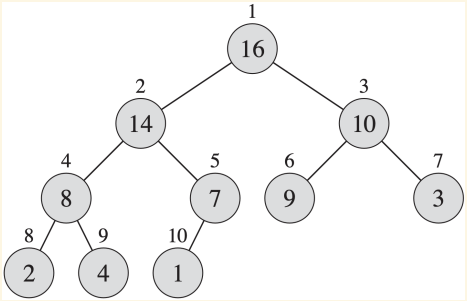
Accessing Parents and Children in the Tree

Here is pseudocode for the procedures for accessing the parent, the left child and the right child of a node i.

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$



Thanks to the PARENT procedure we can express the

- max-heap property as $A[\text{PARENT}(i)] \geq A[i]$.
- min-heap property as $A[\text{PARENT}(i)] \leq A[i]$.

6 / 34

Heap Operations

Overview

The following table shows operations for a max-heap with n elements. Analogous operations also exist for min-heaps.

Operation	Purpose	Worst-case running time
MAX-HEAPIFY	Maintain heap property	$O(\log n)$
BUILD-MAX-HEAP	Produce max-heap from unordered input	$O(n)$
HEAPSORT	Sort array in place	$O(n \log n)$
MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, HEAP-MAXIMUM	Allow heap to implement a priority queue	$O(\log n)$

7 / 34

MAX-HEAPIFY
Pseudocode

Let i be an index of the array A. When MAX-HEAPIFY(A, i) is called, it assumes that the trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but $A[i]$ might be smaller than its children.

MAX-HEAPIFY(A, i)

```
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4        largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7        largest = r
8  if largest ≠ i
9        exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```


8 / 34

MAX-HEAPIFY
Illustration

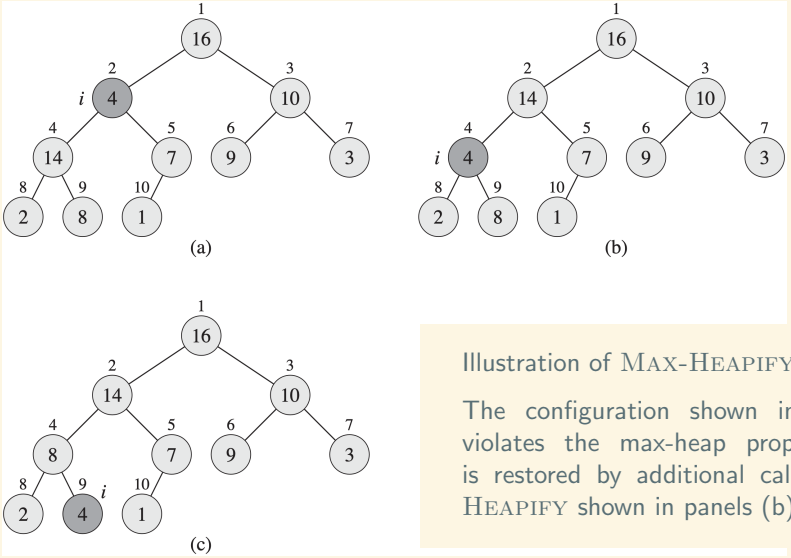


Illustration of MAX-HEAPIFY$(A, 2)$.

The configuration shown in panel (a) violates the max-heap property, which is restored by additional calls to MAX-HEAPIFY shown in panels (b) and (c).

9 / 34

Max-Heapify

Worst-Case Running Time: $O(\log n)$

The running time of Max-Heapify on a subtree of size n rooted at node i is the sum of two contributions:

- $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\textsc{Left}(i)]$ and $A[\textsc{Right}(i)]$
- Time to run Max-Heapify on a subtree rooted at one of the children of node i.
  If i has depth i, then the largest possible ratio of the number of nodes in i's subtree and in one its children's subtree equals $\frac{2^{d+1}-1}{3\cdot 2^d-1}$, which occurs when the bottom level of the tree is exactly half full.

Because $\frac{2^{d+1}-1}{3\cdot 2^d-1} < \frac{2}{3}$, one can show that the running time of Max-Heapify is $O(\log n)$. See section 4.6 in Cormen et al. (2009).

10 / 34

Build-Max-Heap

Pseudocode

The procedure Build-Max-Heap builds a heap from an array A by going through all non-leaf nodes in a bottom-up manner.

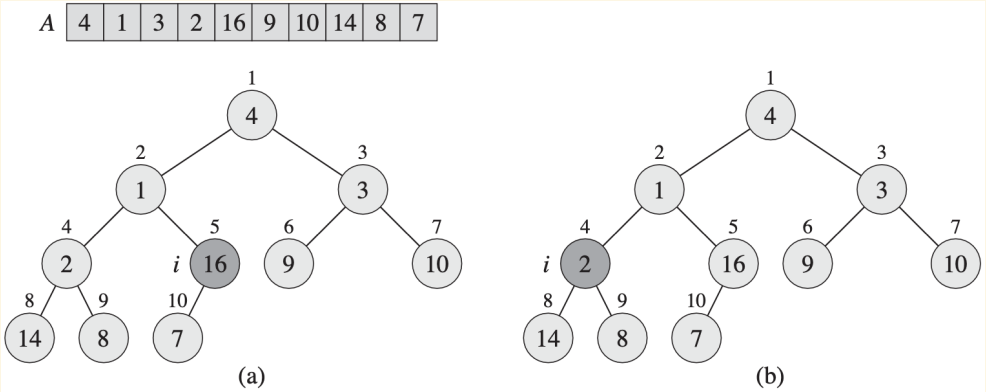
Build-Max-Heap(A)

1 $A.heap\text{-}size = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3 Max-Heapify(A, i)

Before each iteration of the **for** loop, each node $\lfloor A.length/2 \rfloor + 1$, $\lfloor A.length/2 \rfloor + 2, \ldots, n$ is the root of a max-heap. Thus, the condition required for the call of Max-Heapify is satisfied.

11 / 34

BUILD-MAX-HEAP
Illustration (1 of 3)

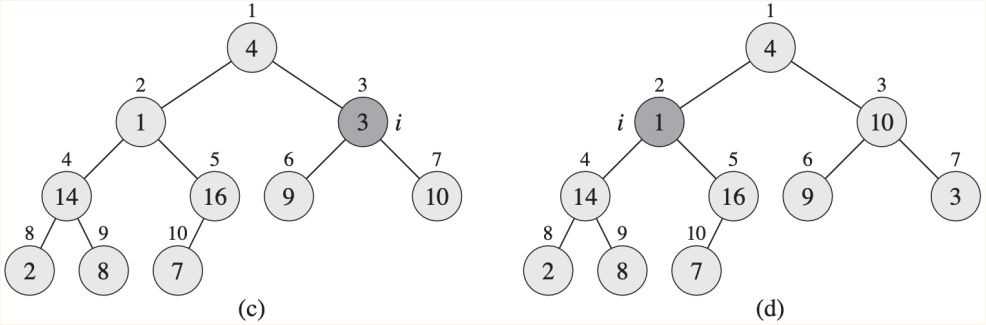


(a) The initial value of the loop index i is 5. The subtree rooted at the node with index i already satisfies the max-heap property.

(b) BUILD-MAX-HEAP decrements i to 4.

12 / 34

Build-Max-Heap
Illustration (2 of 3)

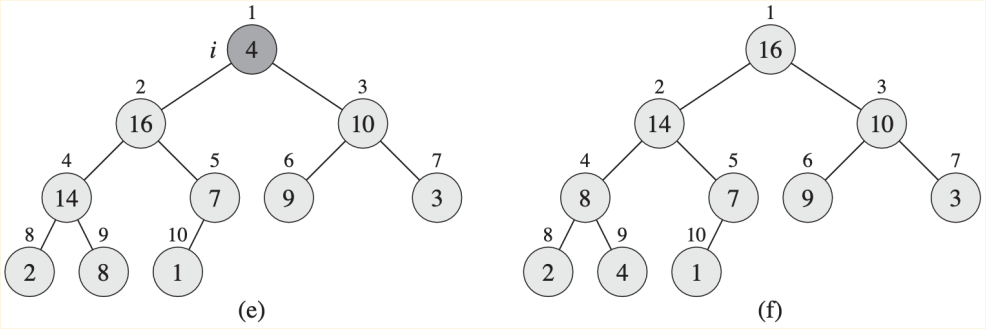


(c) The values 2 and 14 trade places to impose the max-heap property in the subtree rooted at the node with index $i = 4$. Afterwards, i is decremented to 3.

(d) The nodes with the values 3 and 10 trade places before i is decremented to 2.

13 / 34

Build-Max-Heap
Illustration (3 of 3)



(e) To impose the max-heap property in the subtree rooted at node 2, two swaps are necessary: first, the values 1 and 16 trade places, then 1 and 7. The next loop index is $i = 7$.

(f) First, the values 4 and 16 trade places, then 4 and 14, and, finally, 4 and 8. Afterwards, Build-Max-Heap terminates.

14 / 34

Build-Max-Heap

Worst-Case Running Time: $O(n)$

Each call to Max-Heapify costs $O(\log n)$ time, and Build-Max-Heap makes $O(n)$ such calls. Thus, Build-Max-Heap runs in $O(n \log n)$ time.

While this result is correct, it is possible to derive a tighter upper bound for the running time. The key observation is that the time needed by Max-Heapify(A, i) depends on the height of the node i in the tree, and the heights of most nodes are small.

A careful analysis based on this observation reveals that the time required by Build-Max-Heap is only $O(n)$. See section 6.3 in Cormen et al. for details.

15 / 34

Heapsort

Pseudocode

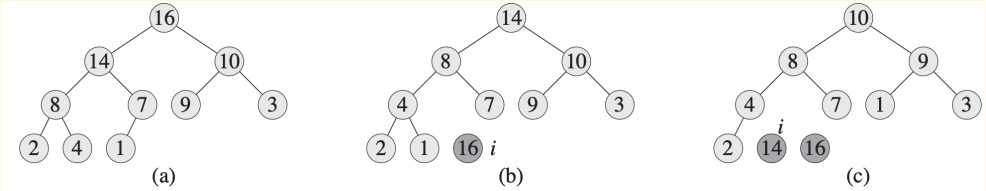
In this course, you have already learnt a variety of sorting algorithms (e.g. insertion sort, quicksort, bubble sort). An alternative to these algorithms is based on heaps.

The heapsort algorithm maintains a max-heap A and iteratively swaps the maximum, which must be $A[1]$, with $A[A.length]$. After each swap, the heap size is reduced by 1, and the max-heap property is restored.

HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 **for** $i = A.length$ **downto** 2
3 exchange $A[1]$ with $A[i]$
4 $A.heap\text{-}size = A.heap\text{-}size - 1$
5 MAX-HEAPIFY($A, 1$)

16 / 34

HEAPSORT
Illustration (1 of 3)

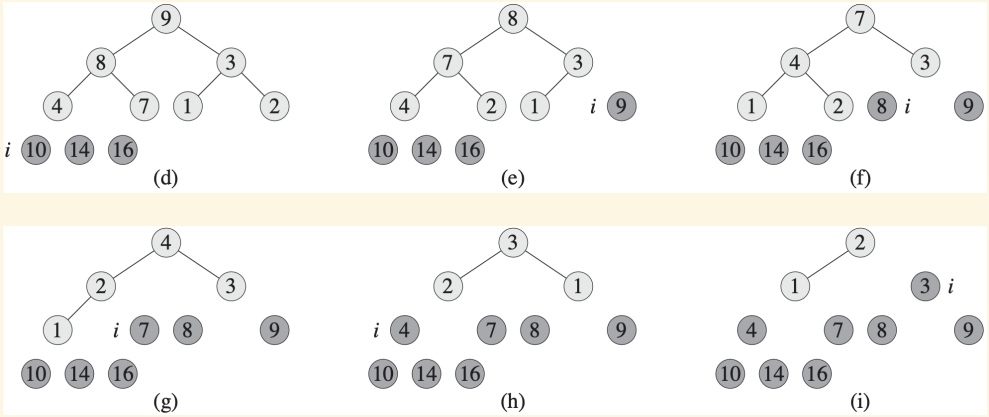


(a) The max-heap structure just after BUILD-MAX-HEAP has built it in line 1.

(b)–(j) The max-heap structure just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap.

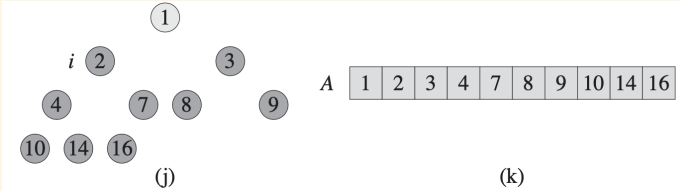
17 / 34

Heapsort
Illustration (2 of 3)



18 / 34

Heapsort
Illustration (3 of 3)



(k) The resulting sorted array A.

19 / 34

Heapsort

Worst-Case Running Time: $O(n \log n)$

Recall the pseudocode of Heapsort:

Heapsort(A)
1 Build-Max-Heap(A)
2 **for** $i = A.length$ **downto** 2
3 exchange $A[1]$ with $A[i]$
4 $A.heap\text{-}size = A.heap\text{-}size - 1$
5 Max-Heapify$(A, 1)$

The call to Build-Max-Heap on line 1 takes $O(n)$ time.

Each of the $n - 1$ calls to Max-Heapify on line 5 takes $O(\log n)$ time.

Therefore, Heapsort takes $O(n \log n)$ time in the worst case. Thus, the growth of the worst-case running time is the same as that of merge sort and tree sort.

20 / 34

Priority Queues

Operations

The heapsort algorithm sorts a single static array. However, heaps can also be used to maintain a dynamic data structure called **priority queue**, which keeps a set S in sorted order after certain types of updates.

Elements in a priority queue possess a **key**. A **max-priority queue** supports the following operations:

- $\textsc{Insert}(S, x)$ inserts the element x into S.
- $\textsc{Maximum}(S)$ returns the element of S with the largest key.
- $\textsc{Extract-Max}(S)$ removes and returns the element of S with the largest key.
- $\textsc{Increase-Key}(S, x, k)$ increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

21 / 34

Priority Queues

Applications

Max-priority queue:

- Job scheduling on a shared computer.
  Items in the queue are jobs, and keys are their urgency (e.g. in Unix measured on a scale from -20 to 19).

Min-priority queue:

- Event-driven simulator.
  Items in the queue are events to be simulated, and keys are their time of occurrence.

22 / 34

Heap-Maximum and Heap-Extract-Max

Pseudocode and Worst-Case Running Times

Heap-Maximum(A)

```
1  return A[1]
```

Worst-case running time of Heap-Maximum is $\Theta(1)$.

Heap-Extract-Max(A)

```
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  Max-Heapify(A, 1)
7  return max
```

The slowest operation in Heap-Extract-Max is Max-Heapify, which has $O(\log n)$ worst-case running time.
Thus, the worst-case running time of Heap-Extract-Max is also $O(\log n)$.

23 / 34

Heap-Increase-Key

Pseudocode and Worst-Case Running Time

Heap-Increase-Key(A, i, key)

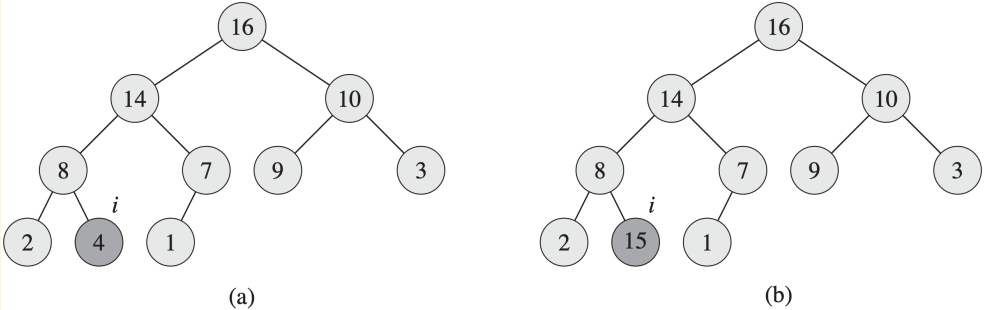
```
1  if key < A[i]
2        error "new key is smaller than the current key"
3  A[i] = key
4  while i > 1 and A[Parent(i)] < A[i]
5        exchange A[i] with A[Parent(i)]
6        i = Parent[i]
```

The number of iterations of the **while** loop is at most equal to the depth of the initial node i, which has an upper bound $O(\log n)$.

Thus, Heap-Increase-Key has a worst-case running time of $O(\log n)$.

24 / 34

Heap-Increase-Key
Illustration (1 of 2)

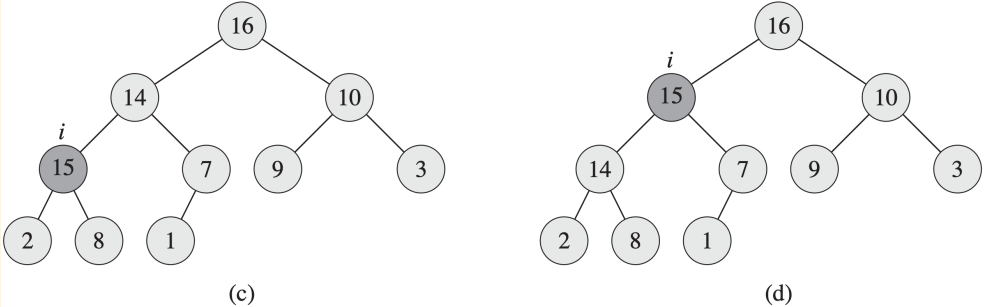


(a) A max-heap, in which the node with index i is highlighted.

(b) This node has its key increased to 15.

25 / 34

Heap-Increase-Key
Illustration (2 of 2)



(c) After one iteration of the **while** loop of lines 4–6, the node and its parents have exchanged keys, and the index i moves up to the parent.

(d) The max-heap after one more iteration of the **while** loop. At this point, $A[\textsc{Parent}(i)] \geq A[i]$. The max-heap property now holds, and the procedure terminates.

26 / 34

Max-Heap-Insert

Pseudocode and Worst-Case Running Time

Max-Heap-Insert(A, key)

1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 Heap-Increase-Key$(A, A[A.heap\text{-}size], key$

Heap-Increase-Key on line 3 is the slowest procedure in Max-Heap-Insert.

Thus, Max-Heap-Insert has a worst-case running time of $O(\log n)$.

27 / 34

Prim's Algorithm

Application of Min-Priority Queues

Prim's algorithm is an alternative to Kruskal's algorithm for finding a minimum spanning tree of a graph.

First, all vertices are inserted into a min-priority queue Q, and all keys are set to ∞. The user supplies a vertex r as an input, which is removed from Q and becomes the root of the minimum spanning tree. The distances between the current tree (consisting only of r at this stage) and all vertices adjacent to the tree (i.e. all neighbours of r) are calculated. The corresponding keys in Q are decreased to be equal to their distance from the tree.

Then, we iteratively remove the minimum key from Q and add the corresponding vertex to the tree.

Prim's algorithm terminates when Q is empty.

28 / 34

Prim's Algorithm

Weights and Node Attributes

We denote the weight of an edge (u, v) as $w(u, v)$. The set of all weights is denoted as w.

Vertices have two attributes:

- $v.key$ is the minimum weight of any edge connecting v to a vertex in the tree.
- $v.\pi$ is the parent of v in the tree.

29 / 34

Prim's Algorithm

Pseudocode

MST-PRIM(G, w, r)

```
1  for each u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = G.V // Use BUILD-MIN-HEAP
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q) // Use HEAP-EXTRACT-MIN
8      for each v ∈ G.Adj[u]
9          if v ∈ Q and w(u, v) < v.key
10             v.π = u
11             v.key = w(u, v) // Use HEAP-DECREASE-KEY
```

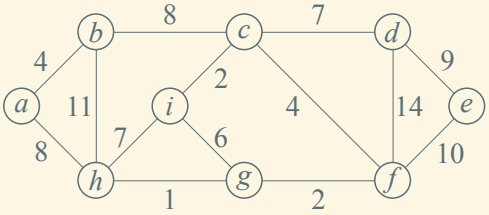

30 / 34

Prim's Algorithm

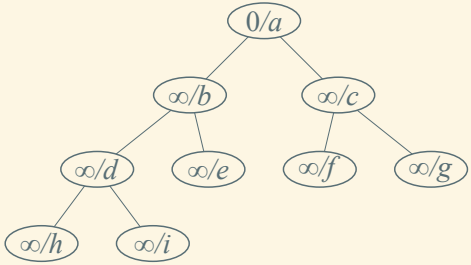
Illustration

The figure depicts the state of G and Q after line 5.

G



Q



The number in each node represents the key. The letter indicates the vertex name.

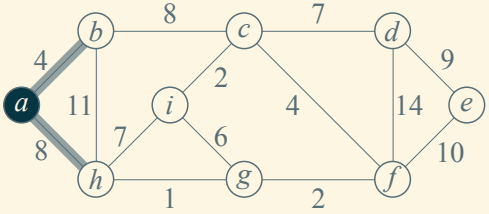
31 / 34

Prim's Algorithm

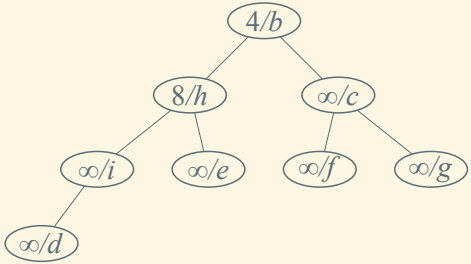
Illustration

The figure depicts the state at the end of an iteration of the **while** loop.

G



Q



Highlighted edges indicate a parent-child relation.

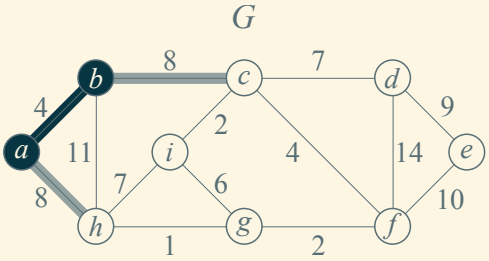
The number in each node represents the key. The letter indicates the vertex name.

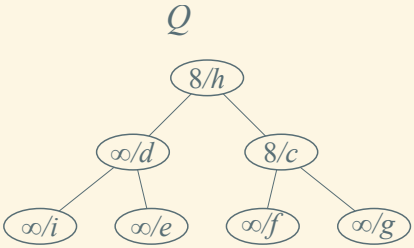
31 / 34

Prim's Algorithm

Illustration

The figure depicts the state at the end of an iteration of the **while** loop.





Highlighted edges indicate a parent-child relation. Black edges are part of a minimum spanning tree.

The number in each node represents the key. The letter indicates the vertex name.

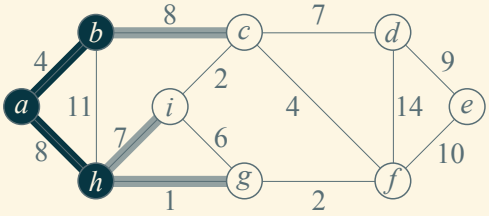
31 / 34

Prim's Algorithm

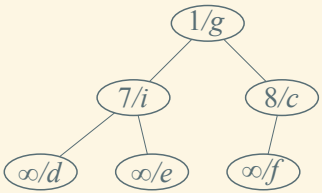
Illustration

The figure depicts the state at the end of an iteration of the **while** loop.

G



Q



Highlighted edges indicate a parent-child relation. Black edges and nodes are part of a minimum spanning tree.

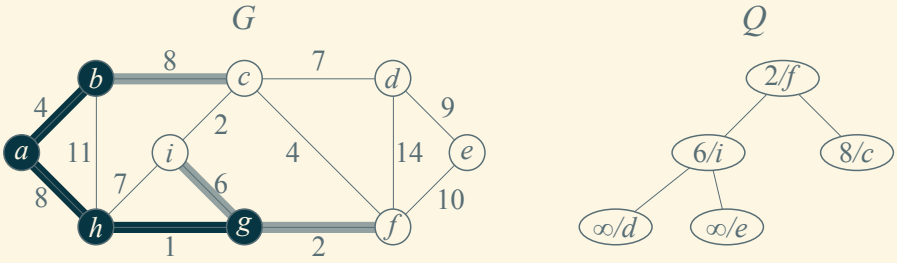
The number in each node represents the key. The letter indicates the vertex name.

31 / 34

Prim's Algorithm

Illustration

The figure depicts the state at the end of an iteration of the **while** loop.



Highlighted edges indicate a parent-child relation. Black edges and nodes are part of a minimum spanning tree.

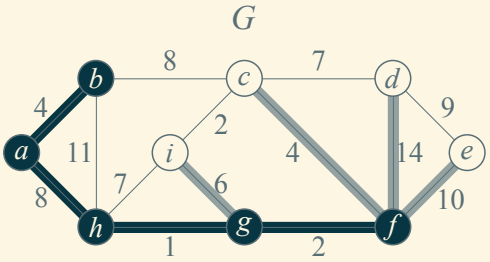
The number in each node represents the key. The letter indicates the vertex name.

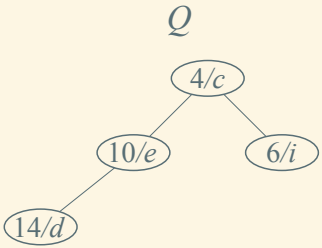
31 / 34

Prim's Algorithm

Illustration

The figure depicts the state at the end of an iteration of the **while** loop.





Highlighted edges indicate a parent-child relation. Black edges and nodes are part of a minimum spanning tree.

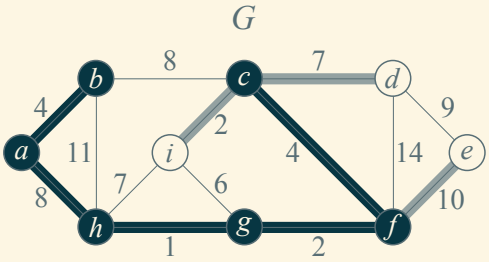
The number in each node represents the key. The letter indicates the vertex name.

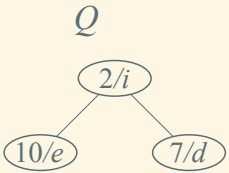
31 / 34

Prim's Algorithm

Illustration

The figure depicts the state at the end of an iteration of the **while** loop.





Highlighted edges indicate a parent-child relation. Black edges and nodes are part of a minimum spanning tree.

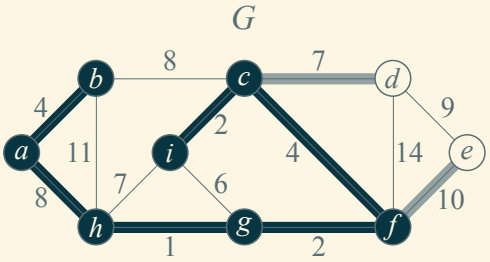
The number in each node represents the key. The letter indicates the vertex name.

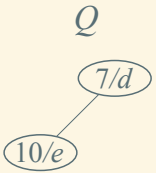
31 / 34

Prim's Algorithm

Illustration

The figure depicts the state at the end of an iteration of the **while** loop.





Highlighted edges indicate a parent-child relation. Black edges and nodes are part of a minimum spanning tree.

The number in each node represents the key. The letter indicates the vertex name.

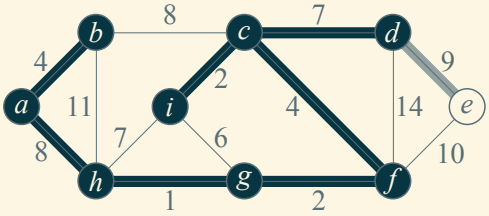
31 / 34

Prim's Algorithm

Illustration

The figure depicts the state at the end of an iteration of the **while** loop.

G



Q

$9/e$

Highlighted edges indicate a parent-child relation. Black edges and nodes are part of a minimum spanning tree.

The number in each node represents the key. The letter indicates the vertex name.

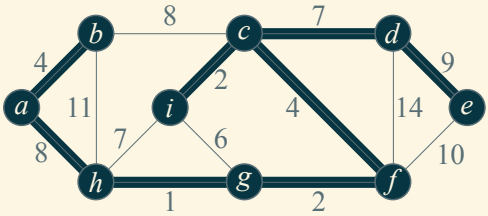
31 / 34

Prim's Algorithm

Illustration

The figure depicts the state at the end of an iteration of the **while** loop. The figure depicts the final state after running MST-PRIM.

G



Q empty

Black edges and nodes are part of a minimum spanning tree.

31 / 34

Prim's Algorithm

Running Time

- BUILD-MIN-HEAP on line 5 requires $O(V)$ time in the worst case.
- There are $|V|$ calls to HEAP-EXTRACT-MIN on line 7. Each call has a worst-case running time of $O(\log V)$. In total, calls to HEAP-EXTRACT-MIN require $O(V \log V)$ in the worst case.
- The **for** loop in lines 8–11 executes $O(E)$ times in total because the sum of the lengths of all the adjacency lists is $2|E|$.
- Within the **for** loop, the check $v \in Q$ can be performed in constant time by maintaining a bit for each vertex that indicates whether it is in Q.
- Each call to HEAP-DECREASE-KEY on line 11 requires up to $O(\log V)$ time.

Thus, the total worst-case running time is $O[(V + E) \log V]$. If the graph is connected, we must have $|V| = O(E)$. In that case, Prim's worst-case running time is $O(E \log V)$.

32 / 34

Implementation Achieving a Better Worst-Case Running Time

Using Fibonacci Heaps Instead of Binary Heaps

It is possible to achieve a slightly improved worst-case running time of $O(E + V \log V)$ by using Fibonacci heaps instead of binary heaps. Details about Fibonacci heaps can be found in Chapter 19 of Cormen et al. (2009).

However, in practice, the constant factors and programming complexity of Fibonacci heaps only make them faster than binary heaps if the number of edges is huge.

33 / 34

Outlook and Conclusion

This Lesson: Prim's Algorithm. Next Lesson: Dijkstra's Algorithm

In this lesson, we studied binary heaps. There are two different types of binary heaps: max-heaps and min-heaps.

Max-heaps are useful for maintaining a dynamic set in descending order as new elements are inserted, the maximum is extracted, and keys are increased. Min-heaps work similarly to maintain a set in ascending order.

Thanks to min-heaps, Prim's algorithm can determine a minimum spanning tree in $O(E \log V)$ time. Thus, the worst-case running time of Prim's algorithm using binary heaps is the same as that of Kruskal's algorithm in our previous lesson.

In the next lesson, we will study Dijkstra's shortest-path algorithm, which can also be implemented efficiently using binary heaps.

34 / 34