Algorithms and Data Structures

Module 2

Lecture 8

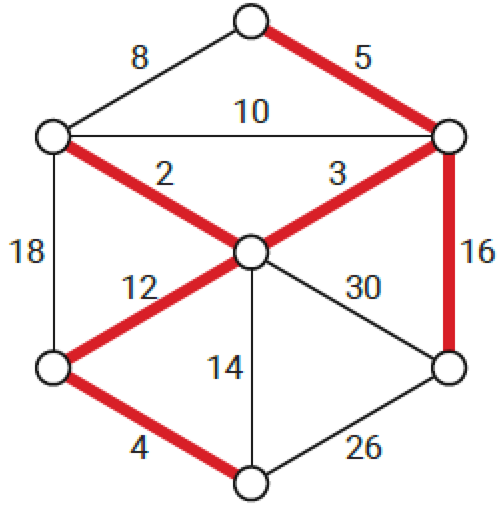
Greedy algorithms.
Minimum Spanning Tree Problem.
Prim’s algorithm.

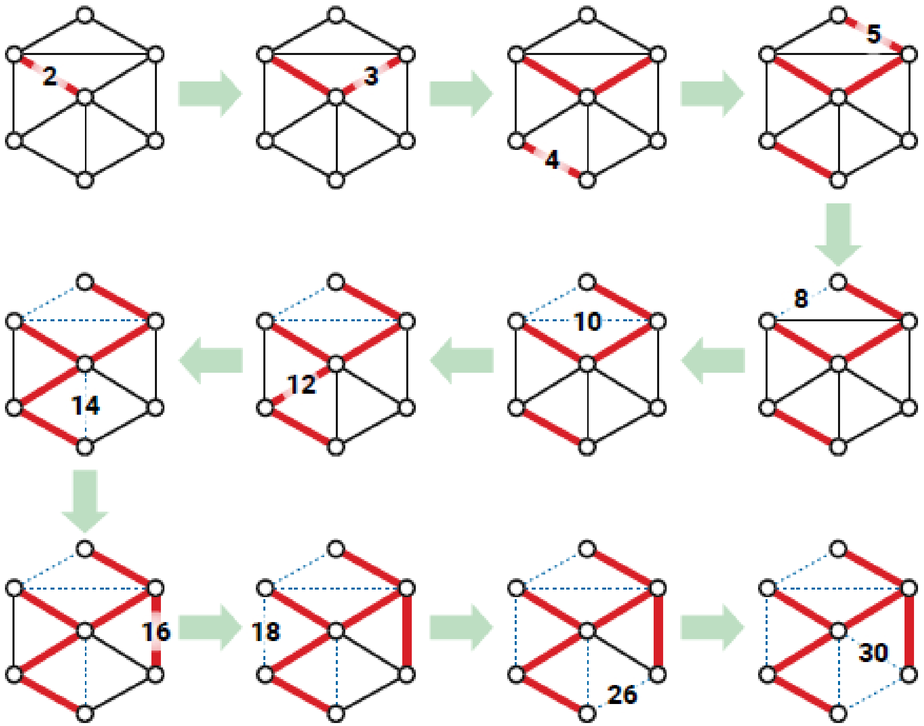
MST: algorithms

A greedy strategy: start with an empty subgraph; add the *lightest* edge such that it does not create a cycle on the subgraph (the lightest *safe* edge).

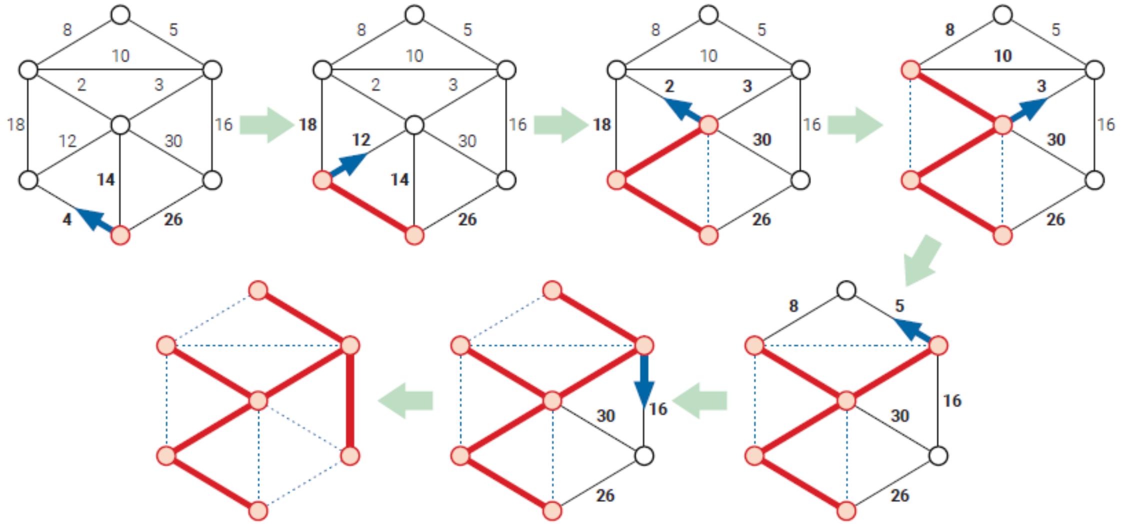
- Kruskal's algorithm: build a *spanning* forest, adding edges until there is one component (tree).
- Prim's algorithm: build the *tree*, adding edges until it spans the graph.

MST: algorithms





Prim's algorithm



Kruskal's algorithm

http://jeffe.cs.illinois.edu/teaching/algorithms/

Prim's algorithm

Given a connected graph $G(V, E)$, $|V| = n, |E| = m$.

1. $T(V_T, E_T)$: $V_T = \{s\}, E_T = \emptyset$
2. Array C[1..*n*], P[1..*n*].
   - $C[s] = 0$; P[s]=*s*.
   - For each $v \in V \backslash \mathrm{V}_T$: $C[v] = w(s, v)$; $P[v] = s$
3. While $V_T \neq V$:
   - Find $v \in V \backslash \mathrm{V}_T$: v has minimum $C[v]$
   - Add v to V_T; add $(P[v], v)$ to E_T
   - Update_C&P(v).

Prim’s algorithm

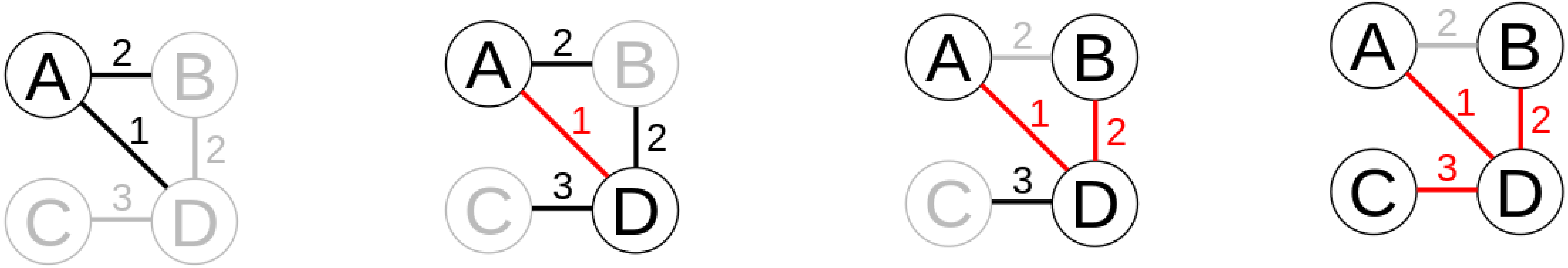
Update_C&P(v)

For each $(v, u) \in E$:

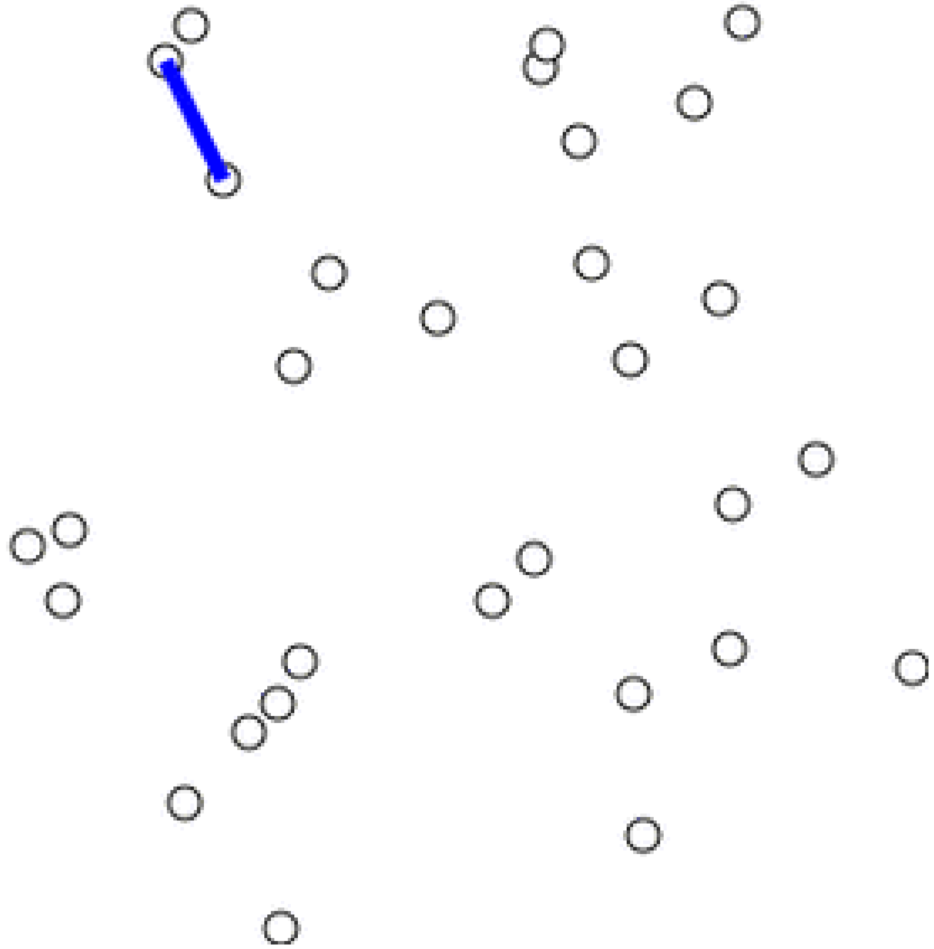
if $u \in V \setminus V_T$ and $C[u] > w(v, u)$:

$C[u] = w(v, u)$

$P[u] = v$



Prim’s algorithm



Prim's algorithm

Given a connected graph $G(V, E)$, $|V| = n, |E| = m$.

1. $T(V_T, E_T)$: $V_T = \{s\}, E_T = \emptyset$
2. Array C[1..*n*], P[1..*n*].
   - $C[s] = 0$; P[1..*n*]=*s*.
   - For each $v \in V \backslash V_T$: $C[v] = w(s, v)$; $P[v] = s$
3. While $V_T \neq V$: *n*-1 iterations
   - Find $v \in V \backslash V_T$: v has minimum $C[v]$???
   - Add v to V_T; add $(P[v], v)$ to E_T O(1)
   - Update_C&P(v). ???

Prim's algorithm

Let us evaluate the total complexity of Update_C&P calls. Actually, we update C[] and P[] at most one time for each edge => the total complexity is $O(m)$.

The complexity of searching for the closest $v \in V \setminus V_T$ depends on the implementation.

Prim’s algorithm

1) Naïve implementation: scan $V \backslash V_T$ and search for the minimum value of $C[v]$. Each scan needs $O(n)$ time => the total time complexity is $O(m + n^2) = O(n^2)$.
2) Use a *priority queue* for keeping $C[v]$ and getting the minimum value at each iteration. The total complexity depends on the priority queue implementation:
   a) Binary heap: $O(m \ \log n)$
   b) Fibonacci heap: $O(m + n \log n)$

Priority queue: definition

- *Priority queue* is an abstract data structure which allows to efficiently append new items and select an item with the highest priority.
- ‘*Priority*’ means numeric values attached to items.
- ‘The highest’ means either ‘the maximum’ or ‘the minimum’ value of priority. Priority queue must be build as either ‘max’ or ‘min’ priority queue; for a max-priority queue one can select an item with the maximum priority and cannot select the minimum priority item, and vice versa.
- Priority queue is not a queue…

Priority queue: definition

Priority queue is an abstract data structure which efficiently implements operations:

- `Init(n)` – initialize an empty priority queue with *n* possible items.
- `Build(S)` – build priority queue containing items of S.
- `Add(x, prior)` – add item *x* with priority *prior* to the priority queue.
- `GetMin()` / `GetMax()` – get the item with the highest priority.
- `DelMin()` / `DelMax()` – delete the item with the highest priority.
- `ChangePriority(x, new_prior)` – change the priority of *x* to *new_prior*.

Priority queue: definition

For Prim's algorithm we apply:

- At the initialization phase:
  - ✓Add(x,prior) – n times
- At the main phase:
  - ✓GetMin() – n times
  - ✓ChangePriority(x,new_priority) – $O(m)$ times.

Priority queue: implementation

We will study and analyze several ways to implement a *priority queue*:

- Array-based implementations
  - ✓Linear (unsorted) array
  - ✓Sorted array
  - ✓Dynamic linked sorted list
- Tree-like data structures
  - ✓Binary search tree
  - ✓2-3 tree
  - ✓Binary heap

Priority queue: array-based implementation

Unsorted array:

- `Add(x, prior)` – append to the end of array. $O(1)$
- `GetMin()` – scan the array for the most prioritized item. $O(n)$
- `DelMin()` – locate the most prioritized item and remove it (shift the tail of the array to the left). $O(n)$
- `ChangePriority(x, new_prior)` – locate item *x* in the array and change its priority. $O(n)$

Total complexity: $O(mn)$

Priority queue: array-based implementation

Sorted array:

- `Add(x, prior)` – insert *x* to the proper position. $O(n)$
- `GetMin()` – get the first item. $O(1)$
- `DelMin()` – delete the first item, shift other items to the left. $O(n)$
- `ChangePriority(x, new_prior)` – locate item *x* in the array, remove it and insert to the new position. $O(n)$

Total complexity: $O(mn)$

Priority queue: array-based implementation

Dynamic linked sorted list:

- `Add(x, prior)` – insert *x* to the proper position. $O(n)$
- `GetMin()` – get the first item. $O(1)$
- `DelMin()` – delete the first item. $O(1)$
- `ChangePriority(x, new_prior)` – locate item *x* in the array, remove it and insert to the new position. $O(n)$

Total complexity: $O(mn)$

Priority queue: binary search tree

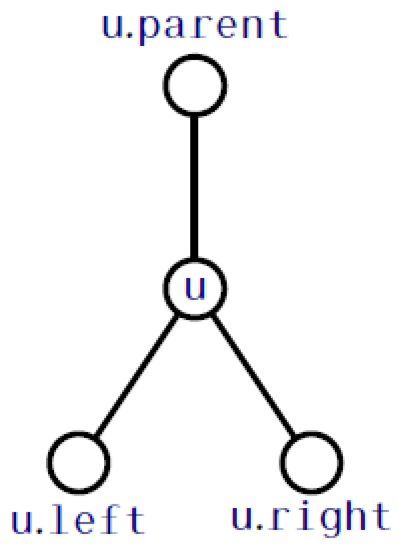
Binary tree is a graph for which the following conditions hold:

a) It is a tree (=connected acyclic graph).

b) One vertex is marked as the *root* of the tree.

c) Each vertex has 0-2 *children*. Vertices with no children are called *leaves*.

d) For each non-leaf vertex, its children are marked as the *left* child and the *right* child. Even if there is only one child, its either the left or the right one.



Height of a binary tree is the maximum length of a path from a leaf to the root.

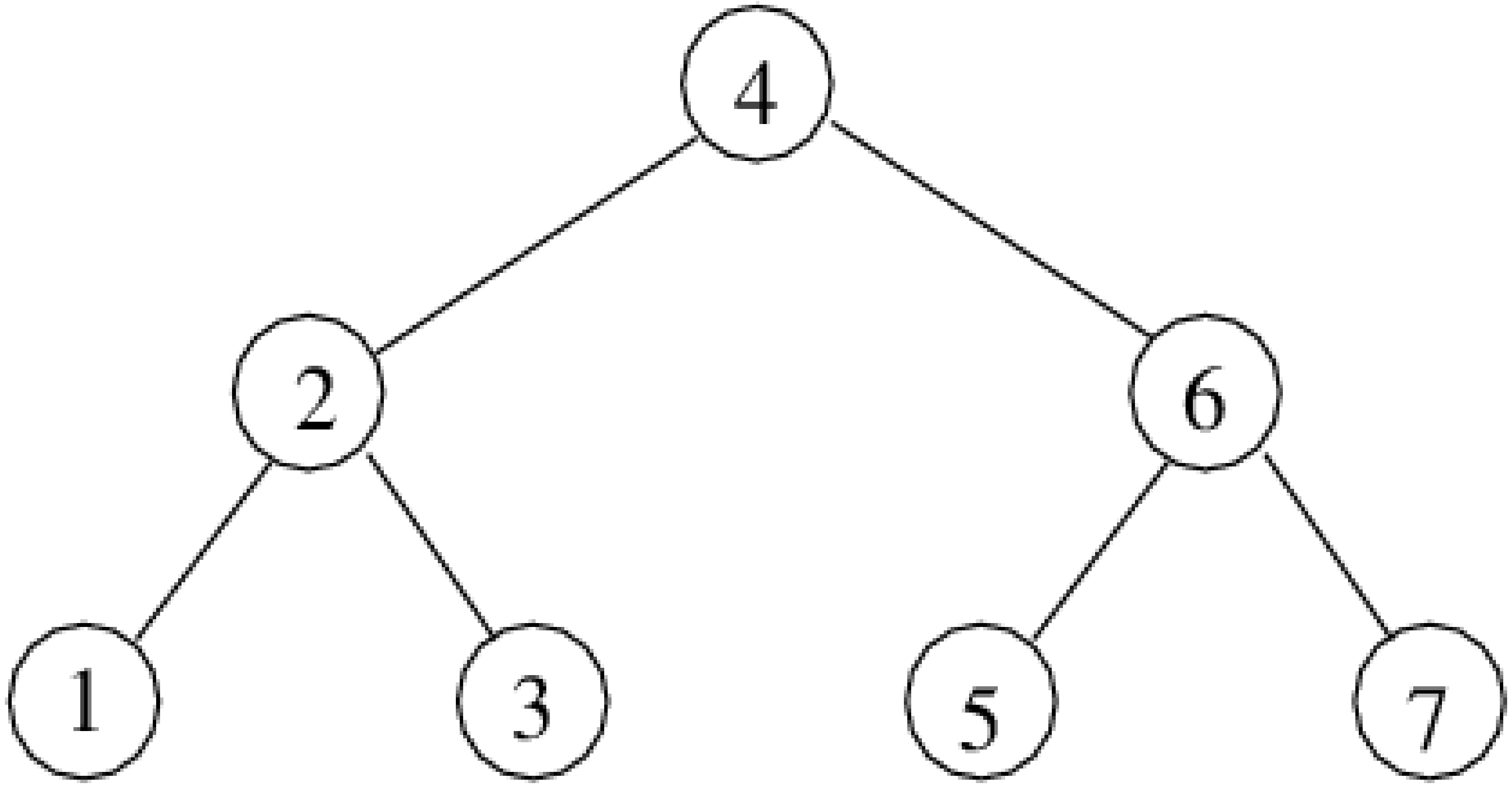
Priority queue: binary search tree

Binary search tree (*BST*) is a binary tree for which the following conditions hold:

a) Each vertex of BST keeps an item with attached numeric key.

b) BST property holds for each vertex with key *K*:
   - All vertices in the left subtree keep keys which are less than K.
   - All vertices in the right subtree keep keys which are greater than or equal to K.

Priority queue: binary search tree



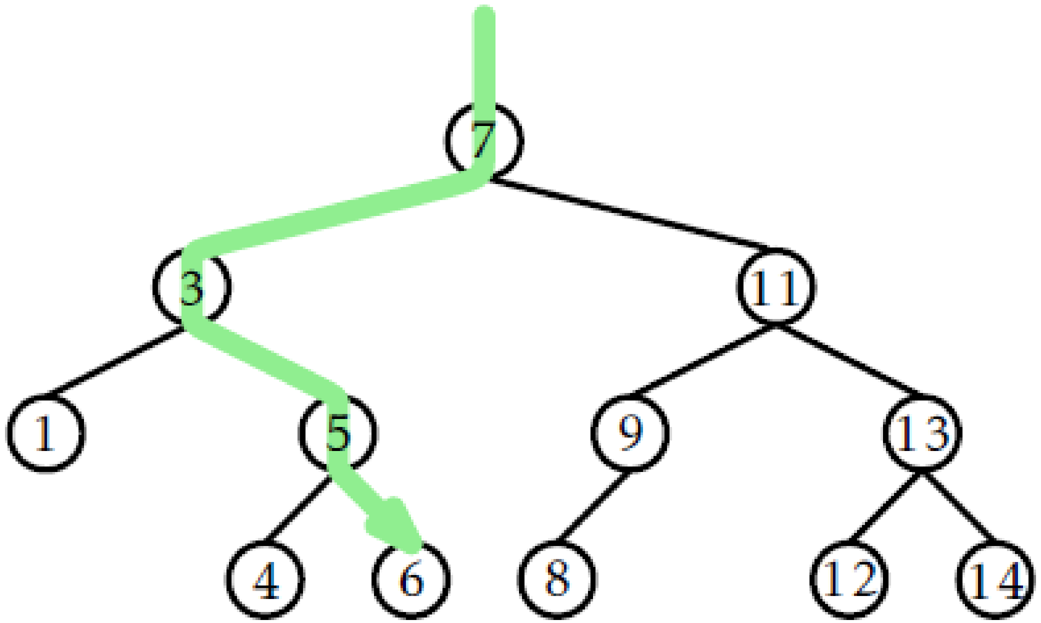
Priority queue: binary search tree

A helper function `Find(K)`:

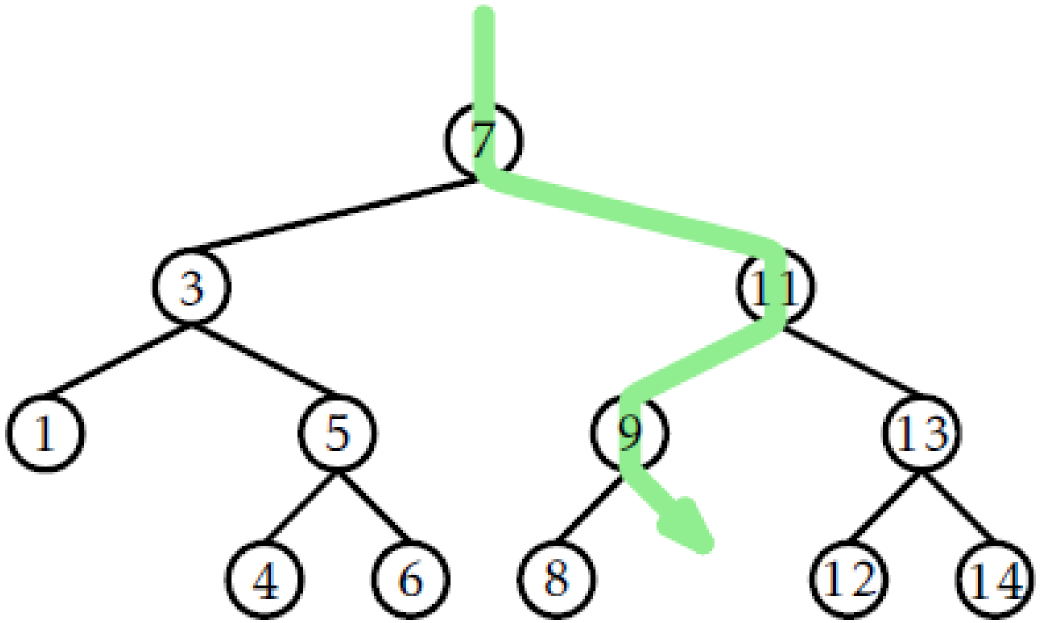
1. Start from the root (current vertex = root of the BST).
2. If current vertex's key = K then key is found.
3. Else if current vertex's key is greater than K then move to the left child (current vertex = left child).
4. Else move to the right child (current vertex = right child).
5. Repeat steps 2-4 until key is found or a leaf is reached.
6. Return 'true' and the position of the found vertex or 'false' and the position where the vertex would be located.

Time complexity: $O(h)$, where h is the height of the BST.

Priority queue: binary search tree



Searching for key=6 (successful)



Searching for key=10 (unsuccessful)

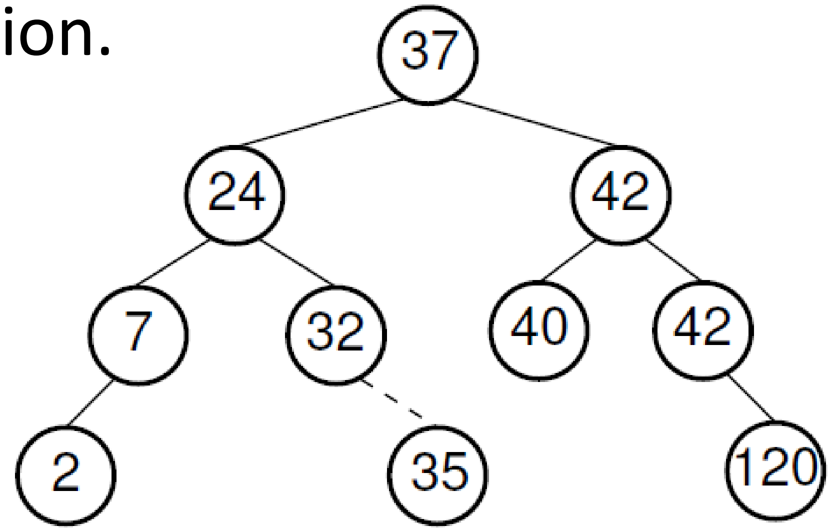
http://opendatastructures.org/

Priority queue: binary search tree

`GetMin()`: start from the root and move to the leftmost vertex, i.e. stop when the current vertex has no left child. Time complexity: $O(h)$.

`Add(x,key)`: search for the position at which *x* would be located in the BST, then add a new vertex to this position.

Time complexity: $O(h)$.

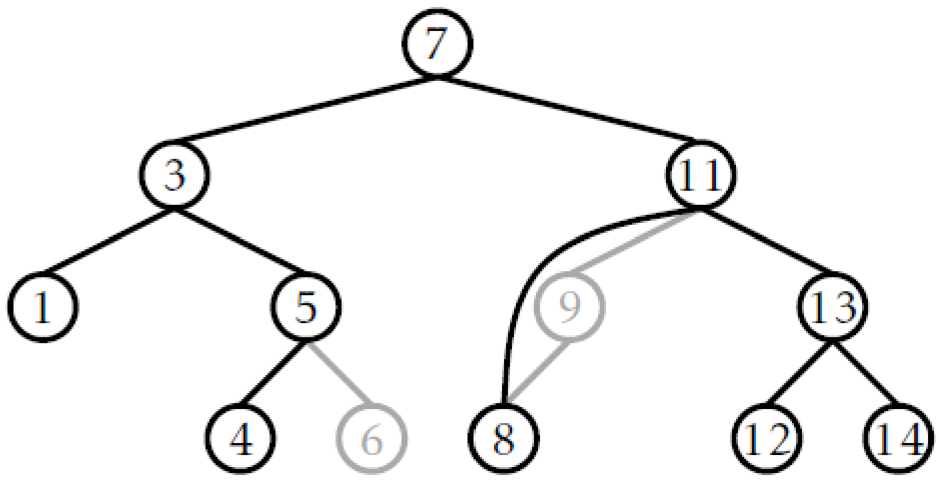


Priority queue: binary search tree

`DelMin()`: delete the leftmost vertex of the BST.

Deleting a vertex v from the BST:

- If v is a leaf: simply remove the vertex, no additional operations needed.
- If v has only one child: replace v with that child.

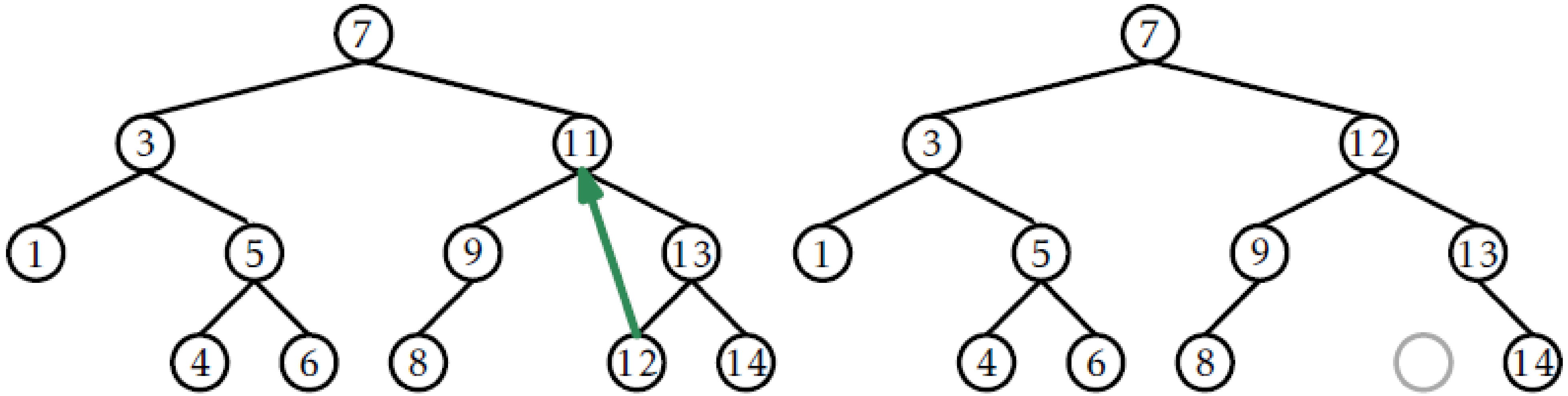


Priority queue: binary search tree

Deleting a vertex v from the BST:

- If v has two children:
  - Find the leftmost vertex w within the right subtree.
  - Move vertex w to the position of v.

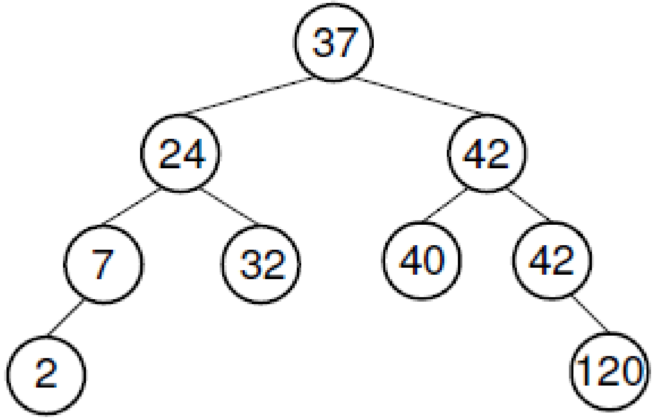
Time complexity: $O(h)$.

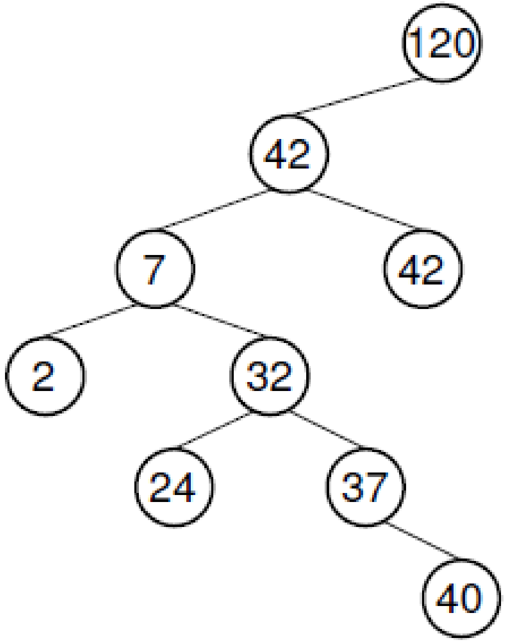


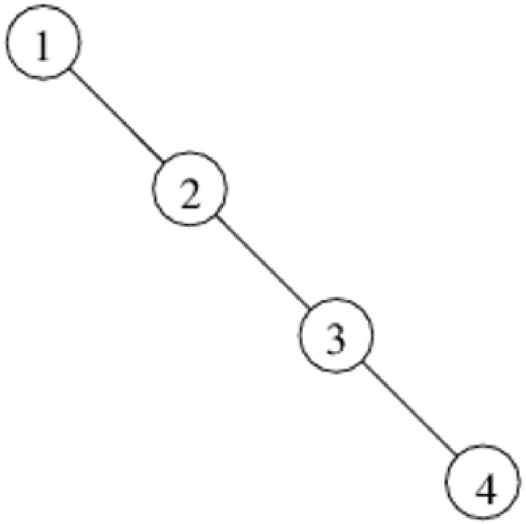
Priority queue: binary search tree

Summary of time complexity for BST: `GetMin`, `DelMin`, `Add` have time complexity $O(h)$, where h is the height of the BST.

Height is $O(\log n)$ on average but $O(n)$ in the worst case ☹







Heaps

A *heap* is a data structure which efficiently implements a priority queue with $O(1)$ time complexity for `GetMin()` and $O(\log n)$ time complexity for `DelMin()`.

Heaps are implemented as tree-based data structures for which all vertices store item+key pairs and the following *heap condition* holds: *the key of any non-root vertex is not less (not greater, for maximizing heaps) than the key of its parent*. Hence the minimum key item is always stored in the root.

Prim's algorithm

Given a connected graph $G(V, E)$, $|V| = n, |E| = m$.

1. $T(V_T, E_T)$: $V_T = \{s\}, E_T = \emptyset$
2. Array C[1..*n*], P[1..*n*].
   - $C[s] = 0$; P[s]=*s*.
   - For each $v \in V \backslash \mathrm{V}_T$: $C[v] = w(s, v)$; $P[v] = s$
3. While $V_T \neq V$:
   - Find $v \in V \backslash \mathrm{V}_T$: v has minimum $C[v]$
   - Add v to V_T; add $(P[v], v)$ to E_T
   - Update_C&P(v).

Prim's algorithm

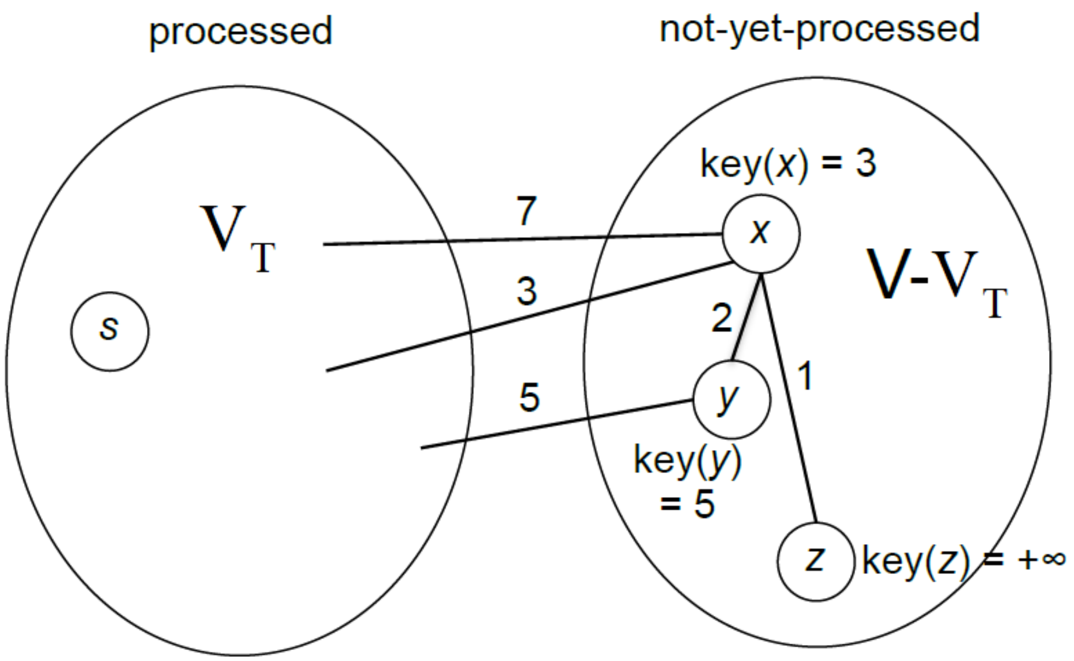
Update_C&P(v)

For each $(v, u) \in E$:

if $u \in V \backslash V_T$ and $C[u] > w(v, u)$:

$C[u] = w(v, u)$

$P[u] = v$



If we use a heap for storing $C[u]$,
the time complexity is $O(m \cdot \log n)$.