Algorithms and Data Structures

Module 2

Lecture 10

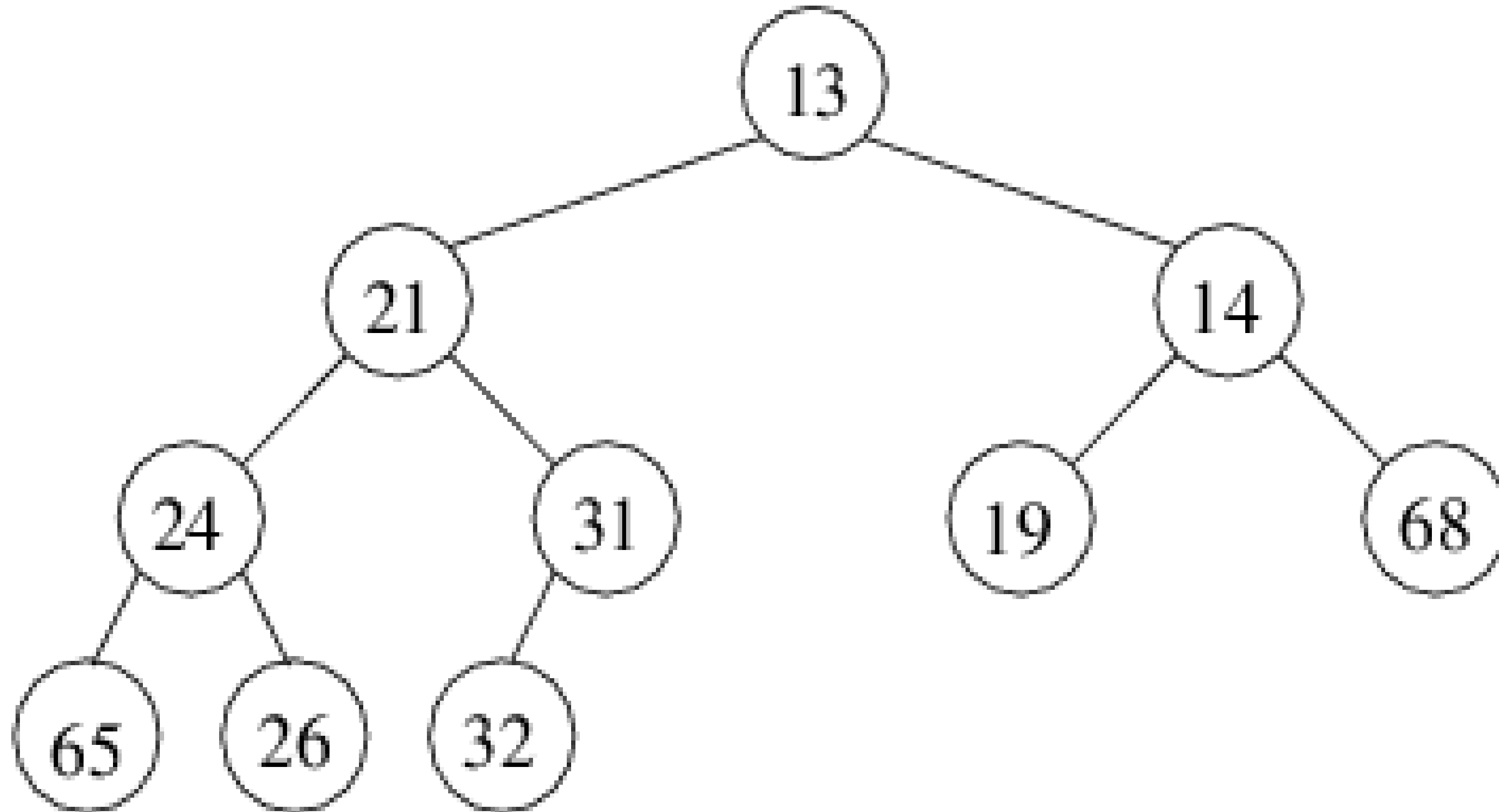
Binary heap implementation and application.

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.

Heaps

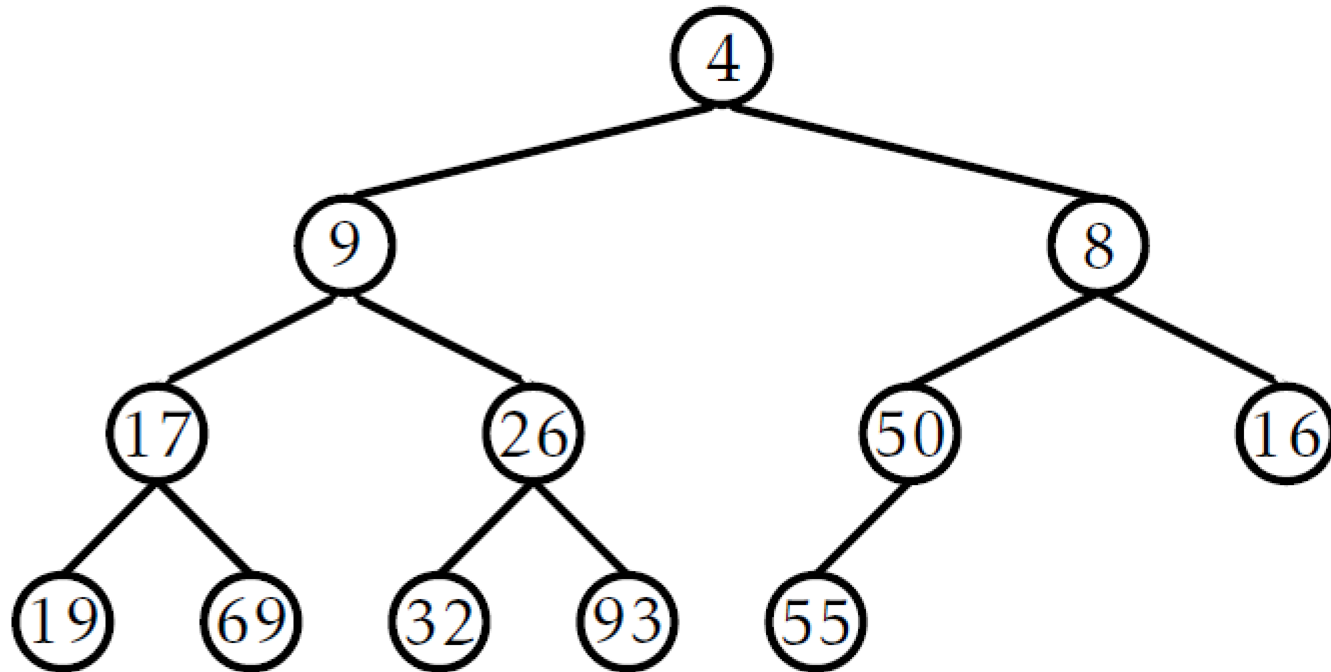


Binary heap

A *binary heap* is implemented as a *complete* binary tree represented as a linear array.

A binary tree is called *complete* iff every *level* of this tree, except possibly the last, is completely filled. If the last level is incomplete, the vertices at the last level are situated as far left as it is possible.

Binary heap

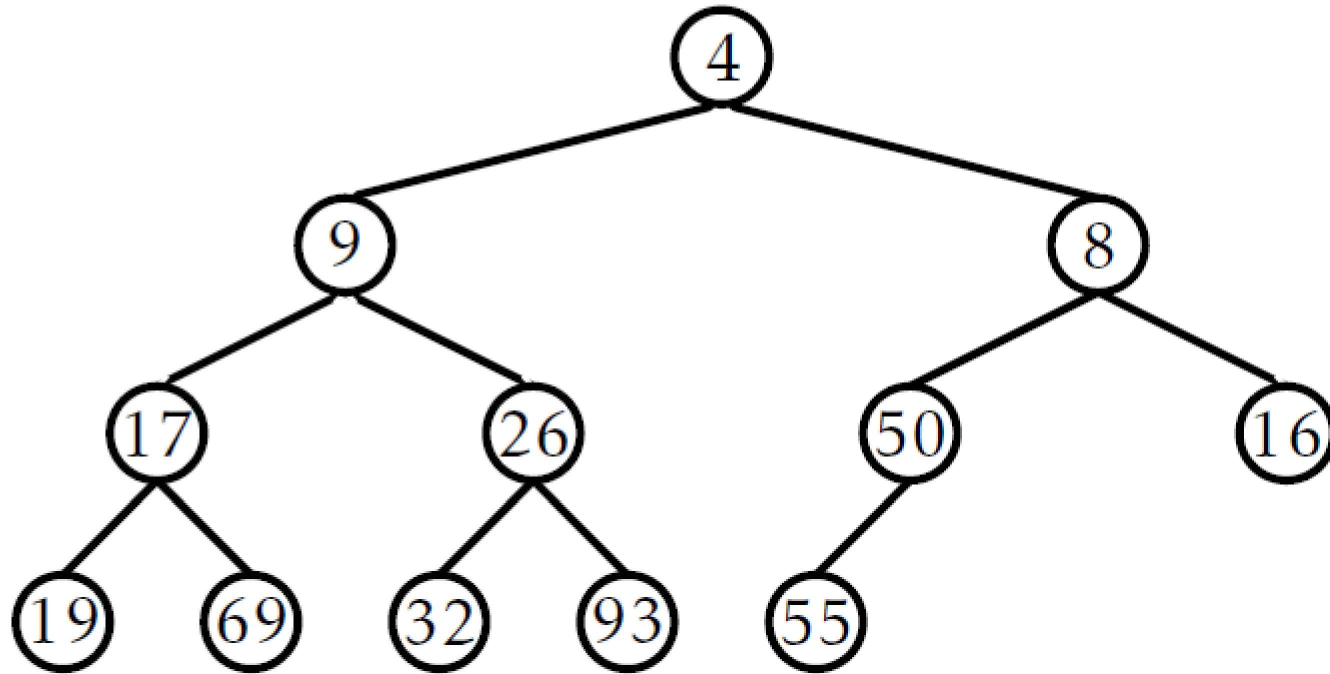


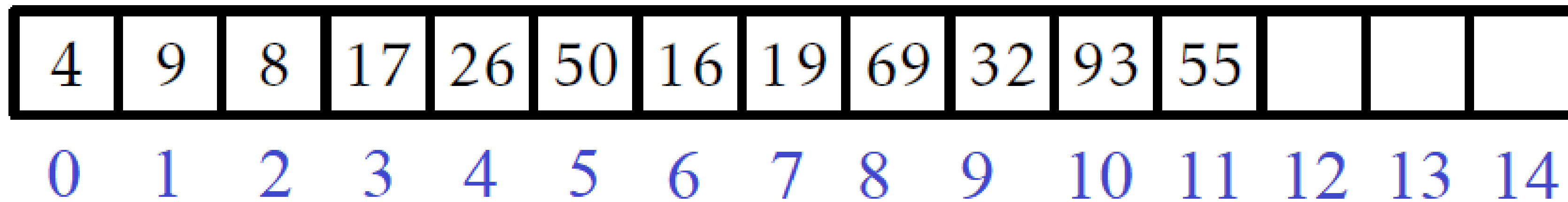
Binary heap

Representing a complete binary tree with a linear array H[0..*n*-1].

- The root is at H[0].
- For any $i < n/2$, the children of $H[i]$ are at $H[2i + 1]$ (the left child) and $H[2i + 2]$ (the right child).

Binary heap





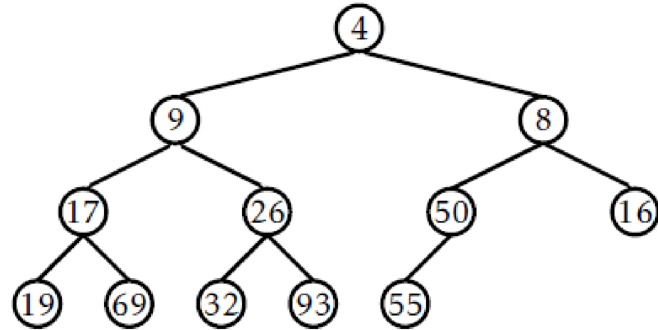
Binary heap: operations

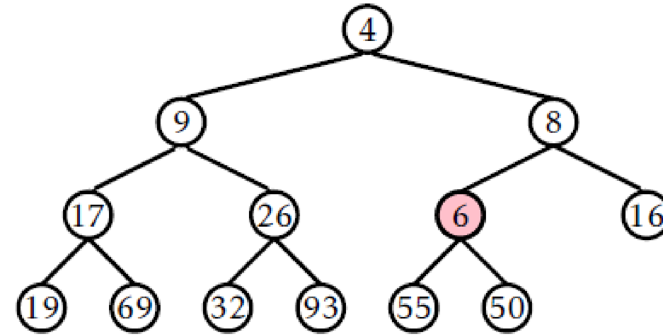
`GetMin()`: return $H[0]$. Time complexity: $O(1)$.

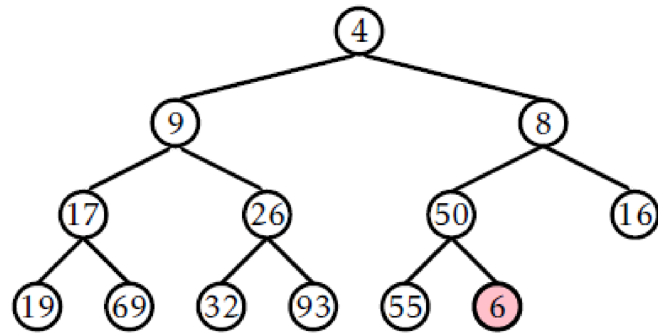
`Add(x, key)`:

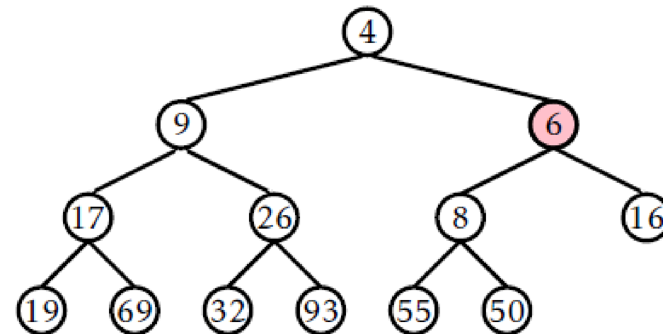
1. Add the new item to the end of H[].
2. Run `SiftUp` operation to make the heap condition hold: if key of the current vertex $H[n-1]$ is less than the key of its parent $H[i]$ $\left(i = \left\lfloor \frac{n}{2} \right\rfloor\right)$ then
   1) Swap $H[n-1]$ with its parent $H[i]$.
   2) SiftUp the vertex $H[i]$.

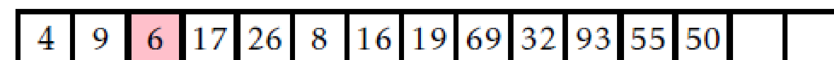
Binary heap: Add (+SiftUp)











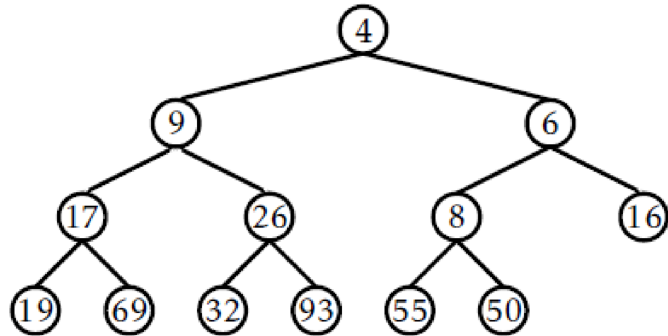
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

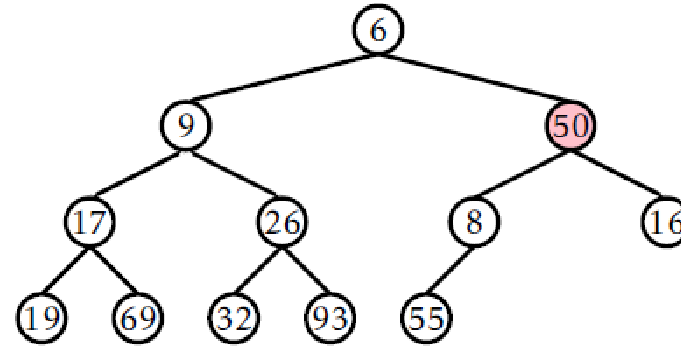
Binary heap: operations

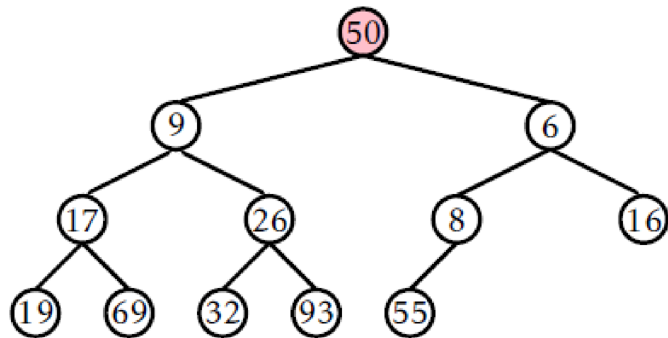
`DelMin()`: delete the root item from the heap:

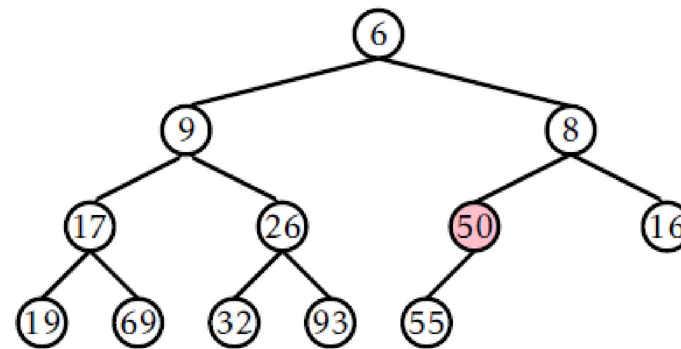
1. Move the item from the last vertex to the root.
2. Starting at the root, recursively run `SiftDown` operation to make the heap condition hold.
   - Check if key of the current vertex $H[k]$ is less than the key of both children. If this holds, stop. Otherwise:
     1) Swap $H[k]$ with its child $H[i]$ with the least key.
     2) SiftDown the vertex $H[i]$.

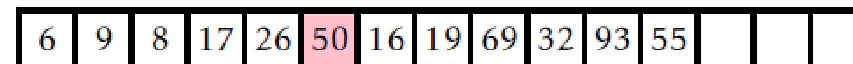
Binary heap: DelMin (+SiftDown)











0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Binary heap: ChangeKey

For efficient implementation of Prim's algorithm, we need one more helper operation: `ChangePriority(x,new_prior)`.

In general, this can be done as a sequence of deleting an item and adding it back with the new key (priority). But for heaps, there is a more efficient implementation:

1. Lookup for `x` in the heap. This does not need traversing the tree, we just keep *direct pointers* from items to the vertices of the heap.
2. Change its key to `new_prior`.
3. If new_prior is less then the previous key, then run `SiftUp` starting from the current vertex. Otherwise, run `SiftDown` starting from the current vertex.

Binary heap: building

For a binary heap, as well as for a priority queue, there may be two versions of initialization procedure:

- `Init(`n`)` – initialize an empty priority queue with *n* possible items.
- `Build(S)` – build priority queue containing items of S.

To build a heap which contains a given set S ($|S| = n$), we can start from an empty heap and add *n* items one after another. For a binary heap, it takes $O(n \log n)$ time. But there is a more efficient way…

Binary heap: building

Given an array H[0..n-1], which contains the items in arbitrary order, we move from the last level to the root, and for each vertex run `SiftDown` procedure.

```
BuildBinaryHeap(H[0..n-1]):
for (i =(n-1)/2; i>=0; i--)
{
      SiftDown(H, i, n)
}
```

This procedure has time complexity $O(n)$.

Binary heap: applications

Sorting an array.

Given: an array $A[0..n-1]$.

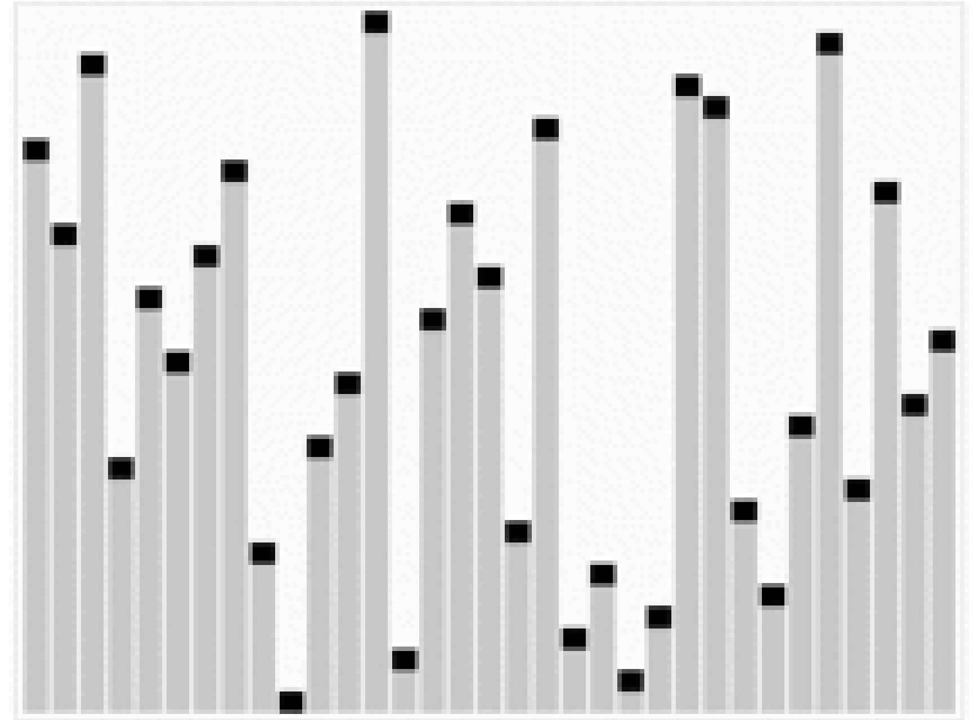
Task: sort A in *ascending* order.

```
HeapSort:
1. Build MaxBinaryHeap(A).        O(n)
2. For i= n-1 downto 0:           O(n log n)
   • Swap A[0] with A[i]
   • SiftDown(A, 0, i-1)
```

Binary heap: applications

<u>Sorting an array</u>.

https://en.wikipedia.org/wiki/Heapsort



A nice animation on YouTube: https://www.youtube.com/watch?v=MtQL_ll5KhQ

Binary heap: applications

Order statistics.

Given: an array $A[0..n-1]$ and an integer k.

Task: get k smallest items (or: the kth smallest item).

```
1. Build MinBinaryHeap(A).
2. For i=1 to k:
   • yield GetMin(); DelMin();
```

$O(n)$
$O(k \log n)$

$O(n + k \log n)$

The specialized order statistics algorithm needs $O(n)$ time.

Binary heap: applications

Prim's algorithm.

To implement Prim's algorithm efficiently, we need a priority queue for storing minimum distances from non-tree vertices to the current tree. At each iteration, we get the closest non-tree vertex and add it to the tree; and then we update the distances

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 $v \in V \backslash V_T$: $C[v] = w(s, v)$; $P[v] = s$
3. While $V_T \neq V$:
   - Find $v \in V \backslash V_T$: v has minimum $C[v]$
   - Add v to V_T; add $(P[v], v)$ to E_T
   - Update_C&P(v).

Prim's algorithm

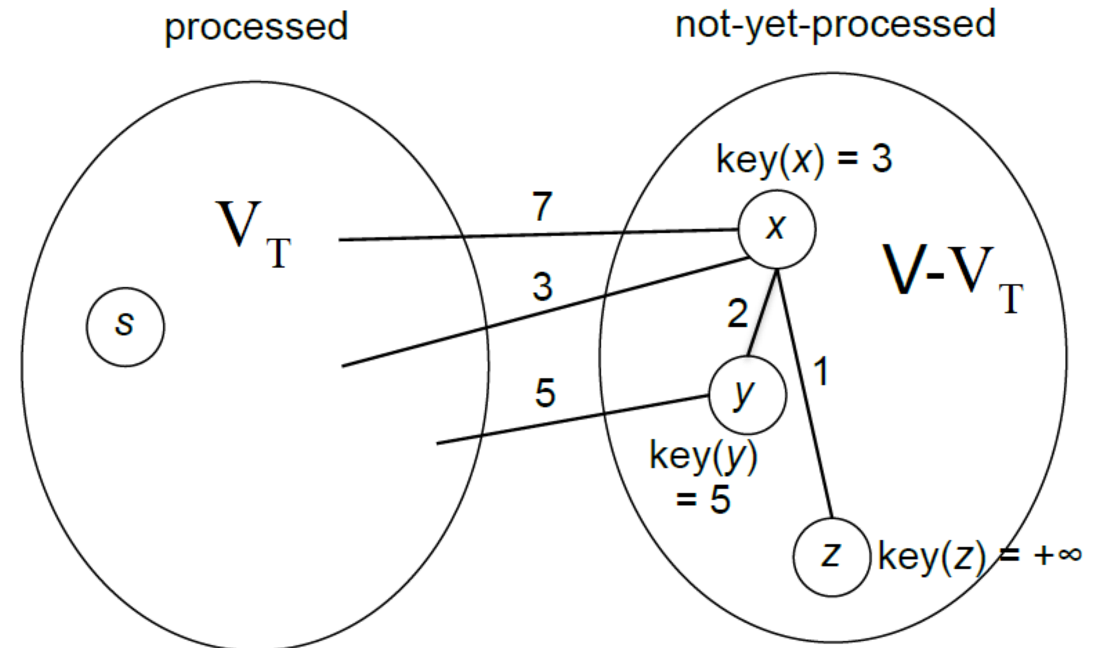
Update_C&P(v)

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 $C[u]$,
we need quick implementations for:

- Accessing a heap vertex for the corresponding graph vertex.
- Updating keys in the heap. `ChangePriority`

Binary heap: ChangeKey (*reminder*)

For efficient implementation of Prim's algorithm, we need one more helper operation: `ChangePriority(x,new_prior)`.

In general, this can be done as a sequence of deleting an item and adding it back with the new key (priority). But for heaps, there is a more efficient implementation:

1. Lookup for `x` in the heap. This does not need traversing the tree, we just keep *direct 'pointers'* from items to the vertices of the heap.
2. Change its key to `new_prior`.
3. If new_prior is less then the previous key, then run `SiftUp` starting from the current vertex. Otherwise, run `SiftDown` starting from the current vertex.

Prim's algorithm

We need quick implementations for:

- Accessing a heap vertex for the corresponding graph vertex.
- Updating keys in the heap.

We use a helper array `Position[]` which for each graph vertex keeps a 'pointer' to the corresponding heap vertex. For array-based implementation of a binary heap, a 'pointer' is just the index within the array `H[]`.

1. When we swap two heap vertices, we also update values in `Position[]`.
2. We use `Position[]` to update distances for vertices not in the tree (within `ChangePriority` function).