Computer Science Department
College of Computer and Information Sciences
King Saud University

CSC 311: Design and Analysis of Algorithms[1]

Dr. Waleed Alsalih

6.1 Graphs [Section 1.4]

A **graph** $G = (V, E)$ is defined by two sets: A set V of **vertices** and a set E of **edges** connecting pairs of vertices. Each edge is defined by a pair of vertices.
An edge can be **directed** or **undirected**. When the pair of vertices defining an edge is unordered, the edge is undirected. In other words, a pair of edges (u, v) is the same as the pair (v, u) (i.e., the direction does not matter). In this case, we say that the vertices u and v are **adjacent** (or connected) to each other by the undirected edge (u, v).
A graph is called undirected if all its edges are undirected.
When the pair of vertices (u, v) is not the same as that of (v, u), the edge (u, v) is directed from u to v (i.e., the direction matters). A graph is called directed if all its edges are directed.
Directed graphs are some times called **digraphs**.
In general, a graph may have a **loop** which is an edge connecting a vertex to itself. However, we will consider graphs with no loops, unless stated otherwise.
For an undirected graph $G = (V, E)$, it is easy to show that:
$0 \leq |E| \leq |V|(|V| - 1)/2$.
A **complete** graph is one in which every pair of vertices is connected by an edge.
A graph with a relatively large number of edges is called **dense**. A graph with a relatively small number of edges is called **sparse**.

[1] This is a summary of the material we cover from the textbook: *Introduction to the Design & Analysis of Algorithms*, A. Levitin, Second Edition, Pearson Addison-Wesley, 2006.

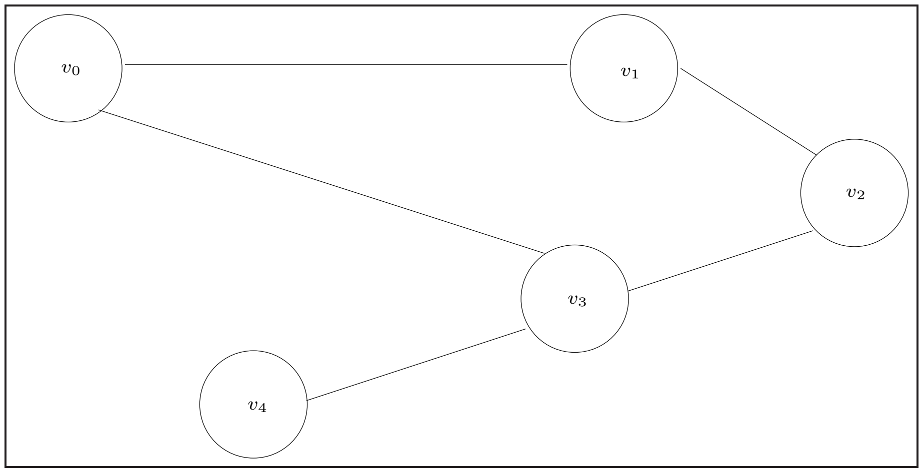

Figure 1: An example of a graph.

Graph representation [Section 1.4]

1. **Adjacency matrix**:
   The adjacency matrix A of a graph $G = (V, E)$ is a $|V|$-by-$|V|$ boolean matrix. Each vertex has one row and one column. $A[i, j] = 1$ if there is an edge from the ith vertex to the jth vertex, and $A[i, j]] = 0$ if there is no such edge. The adjacency matrix of an undirected graph is symmetric (i.e., $A[i, j] = A[j, i]$).

2. **Adjacency lists**:
   The adjacency lists of a graph is a collection of linked lists, one for each vertex. The linked list of a vertex u includes all vertices that share an edge with u.

The adjacency matrix of the graph in Fig. 1 is:

$$\mathbf{M} = \begin{array}{c|ccccc} & v_0 & v_1 & v_2 & v_3 & v_4 \\ v_0 & 0 & 1 & 0 & 1 & 0 \\ v_1 & 1 & 0 & 1 & 0 & 0 \\ v_2 & 0 & 1 & 0 & 1 & 0 \\ v_3 & 1 & 0 & 1 & 0 & 1 \\ v_4 & 0 & 0 & 0 & 1 & 0 \end{array}$$

The adjacency lists of the graph in Fig. 1 are:

$$
\begin{aligned}
L_0 &= \{v_1, v_3\}\\
L_1 &= \{v_0, v_2\}\\
L_2 &= \{v_1, v_3\}\\
L_3 &= \{v_0, v_2, v_4\}\\
L_4 &= \{v_3\}
\end{aligned}
$$

A **weighted graph** is a graph with numbers (weights) assigned to its edges. If a weighted graph is represented using an adjacency matrix, $A[i, j]$ is set to the weight of the edge (i, j). If the edge (i, j) does not exist, $A[i, j] = \infty$. It is convenient some times to put 0's in the diagonal of the adjacency matrix of a weighted graph; this is to reflect the fact that the cost to go from a vertex to itself is 0. To represent a weighted graph using adjacency lists, each node of a linked list carries the id of a vertex and the weight of the corresponding edge.
A **path** from a vertex u to a vertex v is a sequence of adjacent vertices that starts with u and ends with v. The length of a path is the number of edges in it. A path is simple if all of its vertices are distinct. A **cycle** is a path of a positive length that starts and ends at the same vertex.
A graph is **connected** if for any pair of vertices u and v, there is a path from u to v.
A tree is a connected graph with no cycles. For any tree, it is easy to show that
$|E| = |V| - 1$.

Breadth first search [Section 5.2]

Breadth first search proceeds by visiting closer vertices first. We can use it to find the shortest paths from a particular vertex to all other vertices.

The time complexity of this algorithm is $\Theta(|V|^2)$ for the adjacency matrix representation and $\Theta(|V| + |E|)$ for the adjacency list representation.

```
BFS_Single_Source(G(V, E),s)
foreach u ∈ V do
    Status[u]:=unvisited;
    Distance[u]:=∞;
    Path[u]:=null;
end
Status[s]:=visited;
Distance[s]:=0;
Path[s]:=null;
Initialize a queue Q;
Enqueue(Q,s);
while Q is not empty do
    u:=Dequeue(Q);
    foreach v such that (u, v) ∈ E do
        if Status[v] = unvisited then
            Status[v]:=visited;
            Distance[v]:=Distance[u] + 1;
            Path[v]:=u;
            Enqueue(Q,v);
        end
    end
end
```

Computer Science Department
College of Computer and Information Sciences
King Saud University

CSC 311: Design and Analysis of Algorithms[1]
Dr. Waleed Alsalih

6.2 Heaps [Section 6.4]

A heap is a binary tree, with keys assigned to its nodes, that satisfies the folowing two conditions:

1. The binary tree is complete, i.e., all its levels are full except possibly the last level where some rightmost leaves may be missing.
2. For max-heap (min-heap), the key of a node is greater (smaller) than those of its children. We will assume max-heap unless otherwise stated.

Fig. 1 shows an example of a max-heap.
An array can be used to represent a heap as follows. The key of the root is stored in $A[1]$ and $A[0]$ is not used. The children of $A[i]$ are $A[2i]$ and $A[2i+1]$. Leave nodes will occupy the last $\lceil n/2 \rceil$ positions. The heap in Fig. 1 can be represented as follows:[-,100,50,20,40,2,14].

Heap construction

One way to construct a heap of a set of keys is the bottum-up algorithm.

[1] This is a summary of the material we cover from the textbook: *Introduction to the Design & Analysis of Algorithms*, A. Levitin, Second Edition, Pearson Addison-Wesley, 2006.

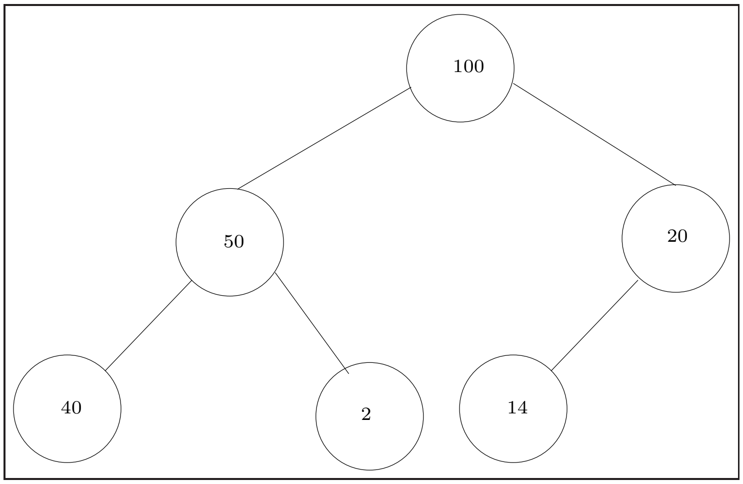

Figure 1: An example of a max-heap.

Worst-case time complexity

Assume, for simplicity, that $n = 2^k - 1$ (i.e., all levels are full). Then, the total number of key comparisons in the worst case is:

$$\sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)) \in O(n) \ ,$$

where h is the height of the tree, i.e., $h = \lfloor \log_2 n \rfloor$.

Exercise:
Prove that

$$\sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)).$$

Hints:

$$\sum_{i=0}^{n} i2^i = 2 + 2^{n+1}(n-1), \text{ and}$$

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1.$$

```
Algorithm HeapBottumUp(H[1..n])
for i = ⌊n/2⌋ downto 1 do
    Sift(H[1..n],i);
end

Algorithm Sift(H[1..n],k)
v:=H[k];
heap:=false;
while not heap and 2k ≤ n do
    j:=2k;
    if j < n then
        if H[j] < H[j + 1] then
            j:=j + 1;
        end
    end
    if v ≥ H[j] then
        heap:=true;
    else
        H[k]:=H[j];
        k:=j;
    end
end
H[k]:=v;
```

Root deletion

Consider an algorithm that removes the maximum key (i.e., the root) from a heap.

Heap sort

Consider an algorithm that sorts an array in a non-decreasing order using heap operations.

```
Algorithm HeapRemoveRoot(H[1..n])
v:=H[1];
H[1]:=H[n];
n:=n − 1;
if n > 0 then
    Sift(H[1..n],1);
end
return v;
```

```
Algorithm HeapSort(A[1..n])
HeapBottumUp(A[1..n]);
for i = 1 to n do
    B[i]=HeapRemoveRoot(A[1..n]);
end
return B[1..n];
```

Time complexity of the heap sort algorithm

This algorithm has two parts: heap construction and a sequence of remove operations. The first part runs in $O(n)$ time. The total number of key comparisons in the second part $C(n)$ meets the following inequality:

$C(n) \le 2\lfloor \log(n-1) \rfloor + 2\lfloor \log(n-2) \rfloor + \cdots + 2\lfloor \log 1 \rfloor \le 2\sum_{i=1}^{n-1} \log i \le 2\sum_{i=1}^{n-1} \log(n-1) = 2(n-1)\log(n-1) \le 2n\log n \in O(n\log n)$.

So the overall complexity of the algorithm is $O(n\log n)$.