Heaps

- complete binary trees with the heap property

- complete: for every $k < \text{(depth of the tree)}$, there are $2^k$ (i.e. maximum possible) nodes at depth $k$. For the last level (equivalent to the depth), the nodes are arranged from left to right with no gaps

- heap property: for every non-leaf node $v$, the key at $v$ is smaller than or equal to the keys of all its children

Heaps are not good for searching since we may need to spend $O(n)$ time.
Heaps

Height of a heap with $n$ keys:

At level $k$: $2^k$ nodes

\[ \forall \ k \in \{0, 1, 2, \ldots, d-1\} \]

At level $d$: between 1 and $2^d$ nodes

Depth $d$:

\[ n \leq 1 + 2 + 4 + \ldots + 2^d = 2^{d+1} - 1 \]

\[ 1 + 2 + 4 + \ldots + 2^{d-1} + 1 \leq n \]

only for complete trees

\[ 2^d \leq n \leq 2^{d+1} - 1 \]

\[ d = \lceil \log n \rceil \]

equal

we'll just use:

\[ \text{depth} = \Theta(\log n) \]
Heaps

Convenient array representation:

Let \( i \) be the index of the current key (e.g. for key 7, \( i = 2 \))

- parent \((i)\) = \( \left\lfloor \frac{i-1}{2} \right\rfloor \) if \( i > 0 \), otherwise parent does not exist (e.g. return -1)
- left-child \((i)\) = \( 2i + 1 \) if \( 2i + 1 < n \), otherwise child does not exist
- right-child \((i)\) = \( 2i + 2 \) if \( 2i + 2 < n \), otherwise child does not exist

root = 0

read the tree (heap) level-by-level, left to right
Priority queues via heaps

- used for implementation of the priority queue data type [see also Section 8.1]

- priority queue: supports insert\( (x) \), remove\text{Min}() 

Can implement priority queues by other data structures

- we choose the best implementation based on how often we perform the two operations \( \text{insert} / \text{removeMin}() \)
  
  → different implementations have different running times

We'll implement the operations via heap in \( O(\log n) \) (both operations)
Priority queues via heaps

How to implement insert(x)?

Suppose the heap array is $A$, we are using $n$ positions of $A$, but $A$ has enough space (size) to fit all our keys.

**Pseudocode:**

1. if $n = A$.length then **MAKE A NEW ARRAY**...
2. $A[n] = x$ // place at the end
3. $n++$
4. $i = n-1$
5. while $(i > 0) \&\&(A[\text{parent}(i)] > A[i])$...
6. $A[\text{parent}(i)] = A[i]$
7. $A[i] = x$
8. $i = \text{parent}(i)$
9. }

inserting: e.g. 25 - easy, at the end
14 - put at the end, switch 14 & 15
3 - put at the end, & "bubble-up"

What is the running time? $O(\text{depth of the heap}) = O(\log n)$
Priority queues via heaps

How to implement removeMin()?

1. \( \text{ret} = A[0] \)
2. \( A[0] = A[n-1] \) // plug the hole
3. \( n-- \)
4. \( i = 0 \)
5. while \((\text{lch}(i) \geq 0 \land \text{A}[\text{lch}(i)] > \text{A}[i] \lor \text{A}[\text{rch}(i)] > \text{A}[i])\) {
   6. if \( \text{A}[\text{lch}(i)] > \text{A}[\text{rch}(i)] \)
      7. \( j = \text{rch}(i) \)
   8. else \( j = \text{lch}(i) \)
      9. if \( j \) is the smaller child
   10. \( \text{tmp} = \text{A}[i] \)
   11. \( \text{A}[i] = \text{A}[j] \)
   12. \( \text{A}[j] = \text{tmp} \)
   13. \( i = j \)
   14. }

What is the running time? \( O(\text{depth}) = O(\log n) \)
HeapSort

- a sort that uses the heap datastructure

```
let B be the input array to sort

HEAPSORT (Input: B)
0. initialize the heap to n=0, allocate A of size B.length
1. for i=0 to B.length-1 do {
2.    INSERT (B[i])
3.    for i=0 to B.length-1 do {
4.        y = REMOVE_MIN()
5.        B[i] = y
6.    }

B = first items
heap
A = inserted items
```

Running time:

$O(n \log n)$
Graphs

Generalization of trees, can include cycles. Formally, $G = (V, E)$ where $V$ = set of nodes, $E \subseteq V \times V$ = set of edges (undirected or directed edges)

$V = \{1, 2, 3, \ldots, 11\}$

$E = \{(1,2), (1,3), (2,3), (2,4), \ldots\}$

Notation: $n = |V|$, $m = |E|$

Notice: $m \leq n^2$

Notice: for trees $m = n - 1$
Graphs

Typical graph properties/terminology:
- path (directed path)
- cycle (tree: a connected acyclic graph)
- subgraph - a subset of vertices & edges
- connected - can get from everybody to everybody else
- connected component
- spanning tree - a subtree that includes all nodes

- distance/length = #edges on the path
- a path from u to v
- a cycle of length 4
- in undirected graphs: cycles always of length \( \geq 3 \)
- a path from u to v
- connected component
Graph representations

- **edge list**
  
  Linked list of all edges
  (possibly plus a list of vertices)
  
  A list of
  
  (1, 2), (1, 4), (1, 3), (3, 0), (3, 4), (4, 0)
  
  Inefficient, e.g. adjacency testing from u to v
  (i.e. there is an edge from u to v) takes \( O(m) \)
  Space \( O(m + n) \)

- **adjacency lists**
  
  For every node we have a list of adjacent nodes (neighboring)
  
  E.g.:
  
  \[
  \begin{aligned}
  0 & \rightarrow x, \text{null} \\
  1 & \rightarrow 2, 4, 3 \\
  2 & \rightarrow x \\
  3 & \rightarrow 4, 0 \\
  4 & \rightarrow 0
  \end{aligned}
  \]
  
  Adjacency testing: \( O(n) \) or even tighter: \( O(\deg(v)) \)
  Space: \( O(m + n) \)
  Node list of all neighbors for a vertex v: \( O(n) \)
  (print)
  Degree: \( \# \) neighbors of v
  Tighter: \( O(\deg(v)) \)

- **adjacency matrix**

  \[
  \begin{array}{cccc}
  0 & 1 & 0 & 0 \\
  1 & 0 & 1 & 1 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 0 \\
  \end{array}
  \]

  Adjacency testing: \( O(1) \)
  Space: \( O(n^2) \)
  List of all neighbors: \( O(n) \)
Graph traversals

- BFS (breadth first search)
- DFS (depth first search)
Graph traversals

BFS (breadth first search) pseudocode

```
assume nodes are: \{0, 1, 2, 3, ... n-1\} = V
0. allocate Q, found of size n = |V|
1. for i = 0 to n do
   2. found[i] = false
   3. dist[i] = \infty
4. }
5. found[s] = true
6. dist[s] = 0
7. Q[0] = s
8. beg = 0
9. end = 1
10. while (beg < end) do
11.     \( \bullet = Q[\text{beg}] \)
12.     for all neighbors \( \Delta \) of \( \bullet \) do
13.         \( a[\text{end}] = \Delta \)
14.         end++
15.         if not found[\( \Delta \)] then
16.             found[\( \Delta \)] = true
17.             dist[\( \Delta \)] = dist[\( \bullet \)] + 1
18. }
```

Running time:

- \( O(n^2) \) if represented by adj. matrix
- \( O(n + m) \) if represented by adj. list

Will keep "working set" (i.e., those that are found but not done)

Overall line 12 iterates \( O(\sum \text{deg}(\text{all nodes})) \) times

= \( O(2m) \) where \( m = \# \text{edges} = |E| \)

\( \boxed{0(n + m)} \)

Works for directed graphs as well

[Section 13.3]
Graph traversals

DFS (breadth first search) pseudocode

(analogous to the green loop on the previous slide)

DFS (Input: a graph G)
1. for i = 0 to n-1 do found[i] = FALSE
2. for s = 0 to n-1 do
3. if not found[s] then DFSs(s)

DFSs (Input: a node s and G)
1. found[s] = TRUE // and print s - corresponds to preorder traversal
2. for every neighbor \( \Delta \) of s do
3. if not found[\( \Delta \)] then
4. DFSs(\( \Delta \))
   // if printing s here, then postorder

Running time: \( O(n+m) \) if adj. lists
\( O(n^2) \) if adj. matrix

Note: many applications will see at least one
Also: works OK for digraphs
- cannot be used to compute distances
Topological sort

Given is a directed acyclic graph, find an order of the nodes such that every edge goes from an earlier node to a later node.

1. find a node with no incoming edges, let it be u
2. place u as the next node
3. remove u and all adjacent edges from the graph
4. continue with step 1 while there are still nodes left

Typical implementation takes O(V+E)

acyclic digraph ⇔ topological sort exists

cyclic digraph ⇒ no top. sort

an example
Topological sort via DFS with time-stamps

**TOPSORT** (Input: a directed graph $G$)
1. for $i=0$ to $n-1$ do $\text{found}[i] = \text{FALSE}$
2. $\text{time} = 0$
3. for $s=0$ to $n-1$ do
4.   **DFS-time-stamps** ($s$)
5. return $\text{order}[]$

**DFS-time-stamps** (Input: a start node $s$)
1. $\text{found}[s] = \text{TRUE}$
2. for every neighbor $u$ of $s$ do
3.   if $\text{found}[u] = \text{FALSE}$ then
4.     **DFS-time-stamps** ($u$)
5. $\text{timestamp}[s] = \text{time}$; $\text{order}[\text{time-1}] = s$
6. $\text{time}++$

Running time: $O(n+m)$ if implemented with adj lists