Queues, Stacks, Graph Traversing
Graph Reminder

- Vertices and edges
- Nodes and arcs
- Representation of graphs:
  -- adjacency matrix
  -- adjacency lists
- Degrees of vertices, indegree, outdegree; regular graphs
- Walks, paths, and cycles; lengths
- Connectivity, connected components
- Trees, root, leaves, parent and child, descendant and ancestor
Graph Traversing

Often we need to visit every vertex of a graph.
There are many ways to do that, two are most usual: breadth first search and depth first search.

In both cases we start with some vertex \( s \).

For BFS:
- visit neighbors of \( s \)
- then visit neighbors of neighbors in turn
- this data structure is called a queue.
Graph Traversing: BFS

Breadth First Search(G,s)
set Discov[s] := true and Discov[v] := false for v ≠ s
Enqueue(Q,s)
while Q is not empty do
    set u := Dequeue(Q)
    for each (u,v) ∈ E do
        if Discov[v] = false then do
            set Discov[v] := true;
            Enqueue(Q,v)
        endif
    endfor
endwhile
Example
Queues Through Array

If we store a queue in an array, we need two pointers to the head and to the tail of the queue.
Enqueue

Enqueue(G,x)
set Q[tail[Q]] := x
if tail[Q] = length[Q] then
    set tail[Q] := 1
else set tail[Q] := tail[Q] + 1

```
7 2 4 6
```

```
3 9 x 6 8 5
```

**Dequeue**

**Dequeue(G)**

set v:=Q[head[Q]]

if head[Q]=length[Q] then
  set head[Q]:=1
else set head[Q]:=head[Q]+1

---

Diagram showing queue operations with arrows indicating head and tail positions.
Graph Traversing: DFS

For DFS:
- start with some vertex s
- visit first neighbor of s
- then visit neighbors of that neighbor
- every time consider the neighbors of the last vertex visited
- this data structure is called a stack
Graph Traversing: DFS
Depth First Search(G,s)

set Explor[s]:=true and Explor[v]:=false for v≠s
Push(S,s)
while not Stack-Empty(S) do
    set u:=Pop(S)
    for each (u,v)∈E do
        if Explor[v]=false then do
            set Explor[v]:=true;
            Push(S,v)
        endif
    endfor
endwhile
Example

![Graph Diagram]

- S
- t
- u
- v
- w
- x
Stacks Through Array

If we store a stack in an array, we need a pointer to the top of the stack.

Stack-Empty(S)
if top[S]=0 then
    return true
else return false

```
8 6 7 2 4 |
  ^
   top
```
Push

Push(S,x)

set top[S]:=top[S]+1
set S[top[S]]:=x
**Pop**

**Pop(S)**

if Stack-Empty(S) then
    error “underflow”
else do
    set top[S]:=top[S]-1
    return S[top[S]+1]
Stacks Through Pointers and Objects

Stacks can also be stored using pointers and objects

Stack-Empty(S)
if top[S]=Nil then
    return true
else return false
Push and Pop

Push(S, x)
set next[x] := top[S]
set top[S] := x

Pop(S)
set t := top[S]
set top[S] := next[top[S]]
return t
Stacks Through Pointers and Objects

Stacks can also be stored using pointers and objects

\[
\text{Stack-Empty}(S) \\
\text{if } \text{top}[S] = \text{Nil} \text{ then} \\
\quad \text{return true} \\
\text{else return false}
\]
Push and Pop

Push(S,x)
set next[x]:=top[S]
set top[S]:=x

Pop(S)
set t:=top[S]
set top[S]:=next[top[S]]
return t
Queues Through Pointers and Objects

Queues can also be stored using pointers and objects

![Diagram showing a queue implemented with pointers and objects]

NIL → tail → head
Enqueue and Dequeue

Enqueue(Q,x)
set next[tail[Q]]:=x
set top[S]:=x

Dequeue(Q)
set x:=head[S]
set head[Q]:=next[head[Q]]
return x
Doubly Linked Lists

To run, say, insertion sort, just a list (or queue, or stack) is not enough, as we need to move along the list back and forth. A **doubly linked** list is used.

Operations:
- List-Search
- List-Insert
- List-Delete
Search

List-Search(L,k)
set x:=head[L]
while x≠NIL and data[x]≠k do
    set x:=next[x]
return x
Insert and Delete

List-Insert(L,x)
set next[x]:=head[L]
if head[L]≠NIL then do
    set prev[head[L]]:=x
set head[L]:=x
set prev[x]:=NIL

List-Delete(L,x)
if prev[x]≠NIL then
    set next[prev[x]]:=next[x]
else
    head[L]:=next[x]
if next[x]≠NIL then
    set prev[next[x]]:=prev[x]
Binary Rooted Trees

If we need to run heap sort on a sequence of numbers of unpredictable length, we cannot organize heap in an array.

A binary tree is used.
Arbitrary Rooted Trees

Sometimes a non-binary tree is needed
Homework

Write pseudocode of Insertion Sort if the data is stored in a doubly linked list

Write pseudocode of Heap Sort using the binary tree representation of data rather than arrays