Overview of this Lecture

✦ A very fast overview of some data structures that we will be using this semester
  ● lists, sets, stacks, queues, networks, trees
  ● a variation on the well known heap data structure
  ● binary search
✦ Illustrated using animation
✦ We are concerned with \( O(\cdot) \) computation counts, and so do not need to get down to C\(^++\)- level (or Java level).
Two standard data structures

Array: a vector: stored consecutively in memory, and typically allocated in advance

This is a singly linked list

first

1 \rightarrow 6 \rightarrow 3 \rightarrow 8 \rightarrow 4 \bullet

cells: hold fields of numbers and pointers to implement lists.
Representations of subsets of a set

subset \( S = \{1, 3, 4, 6, 8\} \)

Array \( A \)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The choice of data structure depends on what operations need to be carried out.
Example 1: Creating an empty set

Initialize: subset $S = \emptyset$

```
1 2 3 4
0 0 0 0 ...
```

$O(n)$ steps

```
0
```

$O(1)$ steps
Example 2: Is $x \in S$?

Is $9 \in S$?

To determine if $9 \in S$, one needs to scan the entire list. The first element is 1, and the list is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. The list is stored in a bit array as follows:

- 1 0 1 1 0 1 0 1 0 0

- O(1) steps

- O(n) steps
Representing a linked list as an array

If Next(j) is empty, then j is not on the list

If Next(j) = ∅, then j is the last element on the list

Array: Next

```
1 2 3 4 5 6 7 8 9 10
```

```
6 8 ∅ 3 4  
```
Two key concepts

**Abstract data types**: a descriptor of the operations that are permitted, e.g.,

- **abstract data type**: set $S$
  - $initialize(S)$: creates an empty set $S$
  - $add(S, s)$: replaces $S$ by $S \cup \{s\}$
  - $delete(S, s)$: replaces $S$ by $S \setminus \{s\}$
  - $IsElement(S, s)$: returns true if $s \in S$
  - etc.

**Data structure**: usually describes the high level implementation of the abstract data types, and can be analyzed for running time.

- doubly linked list, etc
A note on data structures

- Preferences
  - Simplicity
  - Efficiency
  - In case there are multiple good representations, we will choose one
Abstract data type: SetOperations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>initialize(S)</code></td>
<td>$S := \emptyset$</td>
</tr>
<tr>
<td><code>add(S, j)</code></td>
<td>$S := S \cup {j}$</td>
</tr>
<tr>
<td><code>delete(S, j)</code></td>
<td>$S := S \setminus {j}$</td>
</tr>
<tr>
<td><code>IsElement(S, j)</code></td>
<td>returns TRUE if $s \in S$</td>
</tr>
<tr>
<td><code>FindElement(S)</code></td>
<td>if $S \neq \emptyset$, returns $j$ for some $j \in S$</td>
</tr>
<tr>
<td><code>Next(S, j)</code></td>
<td>if $j \in S$, it finds the next element after $j$ on $S$ (viewed as a list)</td>
</tr>
</tbody>
</table>
Implementation using doubly linked list

First = 8

Next( )

Prev

1 2 3 4 5 6 7 8 9 10
Add element 5 to the set (in first position)

Temp := First
First := Elt
Prev(Temp) := Elt
Prev(Elt) := ∅
Next(Elt) := Temp
Add element 5 to the set (in first position)

Adding an element to a set takes $O(1)$ steps using this implementation.
Delete element 2 from the set (assume it is neither in first or last position)
Delete element 2 from the set (assume it is neither in first or last position)

Next(Prev(Elt)) := Next(Elt)
Prev(Next(Elt)) := Prev(Elt)
Prev(Elt) := 0
Next(Elt) := 0

Deleting an element from the set takes $O(1)$ steps using this implementation.
## Operations using doubly linked lists

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize(S)</td>
<td>O(n)</td>
</tr>
<tr>
<td>add(S, j)</td>
<td>O(1)</td>
</tr>
<tr>
<td>delete(S, j)</td>
<td>O(1)</td>
</tr>
<tr>
<td>IsElement(S, j)</td>
<td>O(1)</td>
</tr>
<tr>
<td>FindElement(S)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Next(S, j)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Previous(S, j)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Maintaining disjoint subsets of elements

\[ S_1 = \{8, 2, 4\} \quad S_1 = \{5, 7, 1\} \]
Maintaining ordered lists

The doubly linked list is not efficient for maintaining ordered lists of nodes.

Inserting an element into the set (such as 7) requires finding Prev and Next for 7 (4 and 8), and this requires $O(n)$ time with this implementation.
Complete binary trees for storing ordered lists of nodes or arcs. We assume that the number of nodes is $n$ (e.g., 8) and the number of arcs is $m$.

$S = \{2, 4, 8\}$ \hspace{1cm} n = 8.
Complete binary tree with n elements

Build up the binary tree. In each parent store the least value of its children.

\( S = \{2, 4, 8\} \quad n = 8. \)
Complete binary tree with n elements

Build up the binary tree. In each parent store the least value of its children.

\[ S = \{2, 4, 8\} \quad n = 8. \]
Complete binary tree with n elements

Build up the binary tree. In each parent store the least value of its children.

$S = \{2, 4, 8\}$ n = 8.
Find greatest element less than $j$

e.g., find the greatest element less than 7

O(log n) steps for finding greatest element less than $j$.

start at 7
go up the tree until a node has label < 7. Take left branch.
Choose the largest label child going down.
Delete an element

e.g., delete element 2

O(log n) steps for an deletion

Start at node 2 and update it and its ancestors.

S = \{4, 5, 8\} \quad n = 8.
## Operations using complete binary trees

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>initialize(S)</code></td>
<td>O(n)</td>
</tr>
<tr>
<td><code>add(S, j)</code></td>
<td>O(log n)</td>
</tr>
<tr>
<td><code>delete(S, j)</code></td>
<td>O(log n)</td>
</tr>
<tr>
<td><code>IsElement(S, j)</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>FindElement(S)</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>Next(S, j)</code></td>
<td>O(log n)</td>
</tr>
<tr>
<td><code>Previous(S, j)</code></td>
<td>O(log n)</td>
</tr>
<tr>
<td><code>MinElement(S)</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>MaxElement(S)</code></td>
<td>O(log n)</td>
</tr>
</tbody>
</table>
We can view the arcs of a network as a collection of sets.

Let $A(i)$ be the arcs emanating from node $i$.

e.g., $A(5) = \{ (5,3), (5,4) \}$

Note: these sets are usually static. They stay the same.

Common operations: scanning the list $A(i)$ one arc at a time starting at the first arc.
Storing Arc Lists: $A(i)$

Operations permitted

- Find first arc in $A(i)$
- Store a pointer to the current arc in $A(i)$
- Find the arc after $CurrentArc(i)$

```

i

j  c_{ij}  u_{ij}

1  2  25  30  3  23  35  5  15  40

2  4  15  40  5

3  2  45  10  5  45  60

4

5  3  25  20  4  35  50
```


Scanning the arc list

*CurrentArc(i)* is a pointer to the arc of A(i) that is being scanned or is the next to be scanned.

Initially, *CurrentArc(i)* is the first arc of A(i)

After *CurrentArc(i)* is fully scanned,

*CurrentArc(i) := Next(CurrentArc(i))*
Scanning the arc list

*CurrentArc(i)* is a pointer to the arc of A(i) that is being scanned or is the next to be scanned.

Finding CurrentArc and the arc after CurrentArc takes $O(1)$ steps.

These are also implemented often using arrays called *forward star representations.*
The Adjacency Matrix
(for directed graphs)

A Directed Graph

1 2
a b
c 3
d e

A Directed Graph

• Have a row for each node
• Have a column for each node
• Put a 1 in row i - column j if (i,j) is an arc

What would happen if (4,2) became (2,4)?
The Adjacency Matrix (for undirected graphs)

- Have a row for each node
- Have a column for each node
- Put a 1 in row i - column j if (i,j) is an arc

The **degree** of a node is the number of incident arcs
Adjacency Matrix vs Arc Lists

Adjacency Matrix?
Efficient storage if matrix is very “dense.”
Can determine if \((i,j) \in A(i)\) in \(O(1)\) steps.
Scans arcs in \(A(i)\) in \(O(n)\) steps.

Adjacency lists?
Efficient storage if matrix is “sparse.”
Determining if \((i,j) \in A(i)\) can take \(|A(i)|\) steps.
Can scan all arcs in \(A(i)\) in \(|A(i)|\) steps.
A **tree** is a connected acyclic graph. (Acyclic here, means it has no undirected cycles.)

If a tree has $n$ nodes, it has $n-1$ arcs.

This is an undirected tree.

To store trees efficiently, we hang the tree from a **root node**.

(In principle, any node can be selected for the root.)
A **forest** is an acyclic graph that includes all of the nodes.

A **subtree** of a forest is a connected component of the forest.

To store trees efficiently, each subtree has a **root node**.
One way of storing subtrees

Lists of children

1 → 4 → 2
2
3 → 1 → 6
4
5
6 → 5

node
parent
node
parent (predecessor) array
On storing trees

Trees are important parts of flow algorithms

Some data structures are expressed as trees

The best implementation of trees depends on what operations need to be performed in the abstract data type.
Stacks -- Last In, First Out (LIFO)

Operations:
- create(S) creates an empty stack S
- push(S, j) adds j to the top of the stack
- pop(S) deletes the top element in S
- top(S) returns the top element in S
Queues – First in, First Out (FIFO)

Operations:
- **create(Q)** creates an empty queue Q
- **Insert(Q, j)** adds j to the end of the queue
- **Delete(Q)** deletes the first element in Q
- **first(Q)** returns the top element in S

```
6 5 2 7 3
5 2 7 3
2 7 3
2 7 3 9
```

Delete(Q)
Delete(Q)
Insert(Q,9)
In the ordered list of numbers below, stored in an array, determine whether the number 25 is on the list.
In the ordered list of numbers below, stored in an array, determine whether the number 25 is on the list.
In the ordered list of numbers below, stored in an array, determine whether the number 25 is on the list.

After two more iterations, we will determine that 25 is not on the list, but that 24 and 27 are on the list.

Running time is $O(\log n)$ for running binary search.
Summary

Review of data structures

- Lists, sets, complete binary trees, trees
- Queues, stacks
- Binary search

Next Lecture: Search algorithms