Data Structures
- Elementary data structures
- Heaps
- Binary Search Trees
- Treaps

Elementary data structures
- Arrays and linked lists
- Stacks and queues
- Graphs
- Rooted trees

Arrays

<table>
<thead>
<tr>
<th>x_1</th>
<th>x_2</th>
<th>x_3</th>
<th>x_4</th>
<th>x_5</th>
</tr>
</thead>
</table>

- Read i^{th} cell
- Write i^{th} cell
- Insert i^{th} cell
- Delete i^{th} cell

Stack

- Push(x,S)
- x=Pop(S)

Linked List

- Read i^{th} cell
- Write i^{th} cell
- Insert i^{th} cell
- Delete i^{th} cell

Stack

y_1
y_2
y_3
Stack

Implement with linked lists or arrays to get \( O(\_\_\_) \) per operation:

- \( \text{Push}(x, S) \)
- \( x = \text{Pop}(S) \)

Queue

- \( \text{Enqueue}(x, Q) \)
- \( x = \text{Dequeue}(Q) \)

Queue

Implement with linked list or (circular) array to get \( O(\_\_\_) \) time per operation:

- \( \text{Enqueue}(x, Q) \)
- \( x = \text{Dequeue}(Q) \)

Graph

\( V = \{a,b,c,d\} \)
\( E = \{(a,b),(a,c),(b,c),(c,d)\} \)
### Directed Graph

**V={a,b,c,d}**  
**E={<b,a>,<b,c>,<h,c>,<d,c>}**

### Graph – adjacency list

- **a**: b, c  
- **b**: a, c  
- **c**: a, b, d  
- **d**: c

### Graph – adjacency matrix

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Graphs

- (n vertices, m edges)
  - Is (u,v) an edge of G?  
    - Adjacency list: $O(\_\_\_)$  
    - Adjacency matrix: $O(\_\_\_)$  
  - What are the neighbors of v in G?  
    - Adjacency list: $O(\_\_\_)$  
    - Adjacency matrix: $O(\_\_\_)$

### Trees

- A tree is a connected, acyclic graph.

### Rooted Trees

- A rooted tree is a connected, acyclic graph with one vertex designated as the root.
Rooted Trees
Implement with pointers
- What is the root of T? $O(\_\_\_)$
- What is the parent of v? $O(\_\_\_)$
- What are the children of v? $O(\_\_\_)$

Heap
- Data structure for a set of integers to facilitate __________

Heaps
A heap is a data-structure for storing integer that supports:
1. Build-heap(S): Return a heap on the integers in the set S.
2. Insert(x,H): Insert the integer x into the heap H.
3. Find-min(H): Return the smallest integer in the heap H.
4. Extract-min(H): Remove the smallest integer from the heap H and return it.

Heap: $\{7, 1, 5, 4, 2, 6\}$

1. Rooted, binary tree, filled level by level from the left.
2. (Min) Heap property: the integer stored at a node is no larger than those of its descendents.

Heap
A heap is a data-structure for storing integer that supports:
1. Build-heap(S): Return a heap on the integers in the set S.
2. Insert(x,H): Insert the integer x into the heap H.
3. Find-min(H): Return the smallest integer in the heap H.
4. Extract-min(H): Remove the smallest integer from the heap H and return it.

Insert(3,H) – Step 1 (add)
Insert(3, H) – Step 2 (bubble up)

Insert(3, H) – Step 2 (bubble up)

Insert(3, H) – return

Heap

Insert(3, H) – return

A heap is a data-structure for storing integer that supports:

1. **Build-heap(S)**: Return a heap on the integers in the set S.
   \[ O(n \log n) \]
2. **Insert(x, H)**: Insert the integer x into the heap H.
   \[ O(1) \]
3. **Find-min(H)**: Return the smallest integer in the heap H.
4. **Extract-min(H)**: Remove the smallest integer from the heap H and return it.
Heap

A heap is a data-structure for storing integer that supports:
1. Build-heap(S): Return a heap on the integers in the set S.
2. Insert(x,H): Insert the integer x into the heap H.
3. Find-min(H): Return the smallest integer in the heap H.
4. Extract-min(H): Remove the smallest integer from the heap H and return it.

O(n)
O(1)
O(1)
O(n)

Build-heap{7,1,5,4,2,6}

1. Build-rooted tree
Running Time

The leaves are at height 0. Consider the nodes at height $i$.
How long does it take to fix a subtree rooted at height $i$ assuming its children root heaps? How many nodes are at height $i$? What is the running time of Build-heap?
Array Indexing

Array: A[i]

- ith element of array
- (2i)th element of array
- (2i+1)th element of array

Heap-sort(S)

H = Build-heap(S)

For i = 1 to n

S(i) = Extract-min(H)

Return

Heap-sort is O(n lg n)

O(n)

O(lg n)

Dictionary Data Structure

Data structure that supports add, delete, find for set of keyed records.

- Binary search tree
- Balanced binary search tree
- General search tree
- Hash Table

Binary Search Tree for S

- T is a rooted, binary tree
- Each node in T is assigned a record in S (one-to-one)
- BST Property: For any node X in T
  - If node Y is in the left subtree of X then Y.key ≤ X.key
  - If node Y in the right subtree of X then Y.key ≥ X.key

BST

- Find (x)

If root.key = x return root
If x < root.key recurse on left subtree
If x > root.key recurse on right subtree
Insert (5)

Delete(5)
(Leaf is easy)

Delete(7)
(Node with 1 child is easy)

Delete(8) - Step 1

Delete(8) – Step 2

Operation Run Time

- Search(x) – O(h)
- Insert(x) – O(h)
- Delete(x) – O(h)
Keeping a good balance …

• Search trees: \( O(h) \) time per operation
• “Balanced trees” insure \( O(\log n) \) time per operation.
  – How to balance?
  – When to balance?

Rotations

When to balance?

• Red/black trees
• 2-3 trees
• AVL trees
• Treaps

Makes the when question easy to answer!

Treaps: Step 1

• \( S = \{1,3,5,7,9\} \)
• Each element of \( S \) is assigned a unique “heap key”:
  \( T = (1,15), (3,10), (5,30), (7,0), (9,25) \)

Treaps: Step 2

• \( (1,15), (3,10), (5,30), (7,0), (9,25) \)
• Build tree where
  – \( S \)-key satisfy BST Property
  – \( H \)-key satisfy Heap Property
Treap

\( T=\langle 1,15 \rangle, \langle 3,10 \rangle, \langle 5,30 \rangle, \langle 7,0 \rangle, \langle 9,25 \rangle \)
- Root:
- Left subtree:
- Right subtree:

Treaps

- Claim: If the heap keys are unique then the treap is unique.
- Proof:

Binary Tree Operations

- Insert
- Delete (homework)
Claim: If the heap keys are chosen uniformly at random from [-B,B], where B >> n, then
1. With high probability the keys will be unique.
2. The expected height of the treap is O(lg(n)).