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></th>
<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 · $O(\_\_\_)$
- Write $i^{th}$ cell · $O(\_\_\_)$
- Insert $i^{th}$ cell · $O(\_\_\_)$
- Delete $i^{th}$ cell · $O(\_\_\_)$

Linked List

- Read $i^{th}$ cell · $O(\_\_\_)$
- Write $i^{th}$ cell · $O(\_\_\_)$
- Insert $i^{th}$ cell · $O(\_\_\_)$
- Delete $i^{th}$ cell · $O(\_\_\_)$

Stack

- Push($x, S$)
- $x =$ Pop($S$)
**Stack**

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

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

**Queue**

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

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

\[ V = \{a, b, c, d\} \]
\[ E = \{(b, a), (b, c), (b, c), (d, c)\} \]

Graph - adjacency list

\[ a: b, c \]
\[ b: a, c \]
\[ c: a, b, d \]
\[ d: c \]

Graph - adjacency matrix

\[
\begin{array}{cccc}
  a & b & c & d \\
  a & 0 & 1 & 1 & 0 \\
  b & 1 & 0 & 1 & 0 \\
  c & 1 & 1 & 0 & 1 \\
  d & 0 & 0 & 1 & 0 \\
\end{array}
\]

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
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.

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

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.
   \(O(lg n)\)
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.
A heap is a data-structure for storing integer that supports:

1. **Build-heap(S)**: Return a heap on the integers in the set $S$.
   $O(n)$

2. **Insert(x,H)**: Insert the integer $x$ into the heap $H$.
   $O(\log n)$

3. **Find-min(H)**: Return the smallest integer in the heap $H$.
   $O(1)$

4. **Extract-min(H)**: Remove the smallest integer from the heap $H$ and return it.
   $O(\log n)$

**Heap**

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

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

Fix subtrees

Heap property holds at leaves

Heap

Heap

Bubble down

Heap

Bubble down

Heap

Heap

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?

Implementing a heap in an array

2 4 3 6 5 7
Array Indexing

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)

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

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 and where to balance

How: Rotations

When to balance?

- Red/black trees
- 2-3 trees
- AVL trees
- Treaps
  - Makes the when & where question easy to answer!

Red-Black Trees

- Binary Search Tree that satisfies the following
  - Each node is colored either red or black
  - Every leaf is NIL and black
  - If a node is red each of its children are black
  - For a node $x$, every simple path from $x$ to a leaf has the same number of black nodes

Color this to be a Red-Black Tree
Red-Black Tree

Claim
A Red-Black tree with n internal nodes has height at most 2 \( \lg (n+1) \).

Red-Black Tree: Insert 2

Red-Black Tree: Insert 10

Red-Black Fix

Red-Black Fix

Recolor
Red-Black Fix

1. Rotate left
2. Rotate right & recolor

YUCK

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

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

T = (1, 15), (3, 10), (5, 30), (7, 0), (9, 25)

Binary Tree Operations

• Insert
• Delete

Treaps

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

• Proof:

Treap: Insert (6, -10)

Treap: Rotate
Treap: Rotate

(7,0)  
(6,10)  
(1,15)  
(5,30)  
(9,25)  
(6,-10)

Treap: Rotate

(6,10)  
(3,10)  
(1,15)  
(5,30)  
(9,25)  
(7,0)

Treaps

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 $O(\log(n))$. 