# CS140: Algorithms

Z Sweedyk
Lecture 3
1/25/01



## Data Structures

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

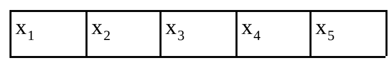


## Elementary data structures

- Arrays and linked lists
- Stacks and queues
- Graphs
- Rooted trees



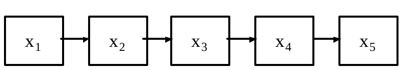
## Arrays




- Read $i^{th}$ cell — O( ___ )
- Write $i^{th}$ cell — O( ___ )
- Insert $i^{th}$ cell — O( ___ )
- Delete $i^{th}$ cell — O( ___ )



## Linked List

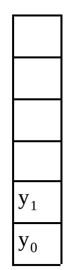



- Read $i^{th}$ cell — O( ___ )
- Write $i^{th}$ cell — O( ___ )
- Insert $i^{th}$ cell — O( ___ )
- Delete $i^{th}$ cell — O( ___ )

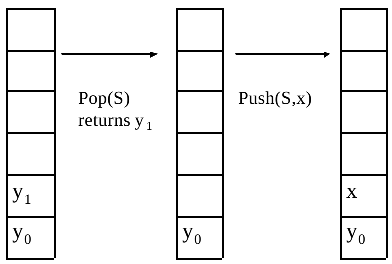


## Stack

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



# Stack





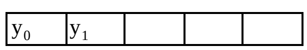

# Stack

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

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



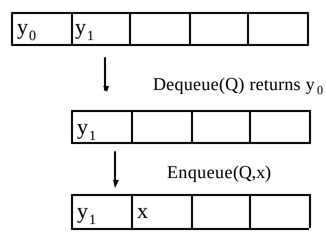
# Queue




- Enqueue(x,Q)
- x=Dequeue(Q)



# Queue






# Queue

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

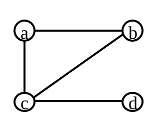
- Enqueue(x,Q)
- x=Dequeue(Q)



# Graph

V={a,b,c,d}

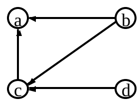
E={(a,b),(a,c),(b,c),(c,d)}



# Directed Graph

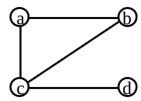
V={a,b,c,d}

E={<b,a>,<b,c>,<b,c>,<d,c>}






# Graph – adjacency list




a: b,c

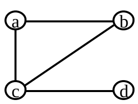
b: a,c

c: a,b,d

d: c



# Graph – adjacency matrix




| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | 1 | 1 | 0 |
| b | 1 | 0 | 1 | 0 |
| c | 1 | 1 | 0 | 1 |
| d | 0 | 0 | 1 | 0 |



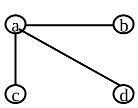
# Graphs
## (n vertices, m edges)

- 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

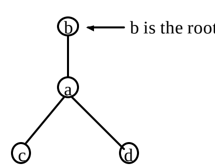
- A tree is a connected, acyclic graph.






# 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 ______________________



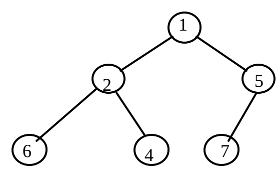
## Heaps

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.



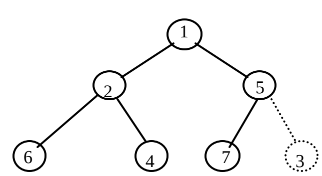
## Heap

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

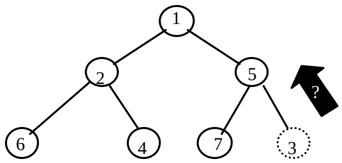
1. Build-heap(S): Return a heap on the integers in the set S.
2. Insert(x,H): Insert the integer x into the heap H.
3. Find-min(H): Return the smallest integer in the heap H. O(1)
4. Extract-min(H): Remove the smallest integer from the heap H and return it.



## Insert(3,H) – Step 1 (add)

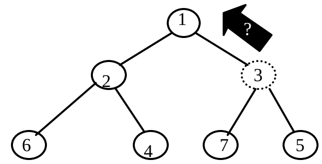


## Insert(3,H) – Step 2 (bubble up)



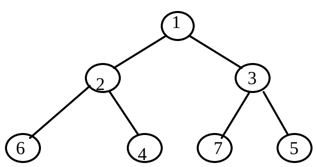



## Insert(3,H) – Step 2 (bubble up)






## Insert(3,H) – return





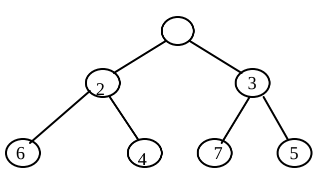

## Heap

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. O(lg 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.

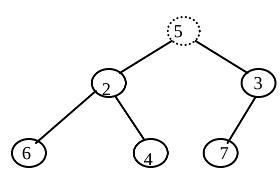


## Extract-min(H) – Step 1 (remove)

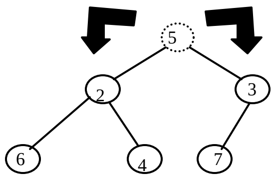





## Extract-min(H) – Step 2 (move last to root)

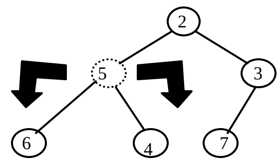


## Extract-min(H) – Step 3 (bubble down)



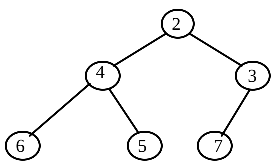



## Extract-min(H) – Step 3 (bubble down)






## Extract-min(H) – return






## 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. O(lg n) Insert(x,H): Insert the integer x into the heap H.
3. O(1) Find-min(H): Return the smallest integer in the heap H.
4. O(lg n) Extract-min(H): Remove the smallest integer from the heap H and return it.



## Heap

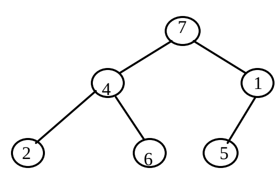
A heap is a data-structure for storing integer that supports:

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



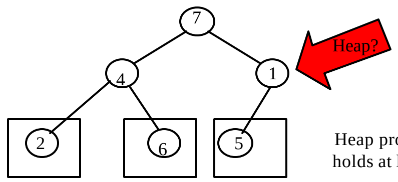
## Build-heap{7,1,5,4,2,6}

Build rooted tree



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

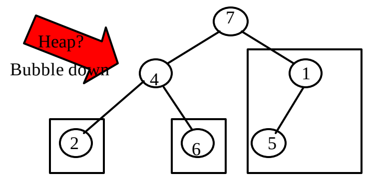
Fix subtrees




Heap property holds at leaves

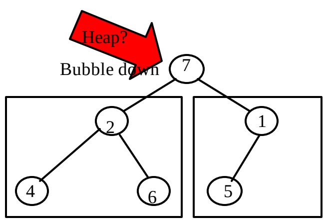


## Build-heap{7,1,5,4,2,6}



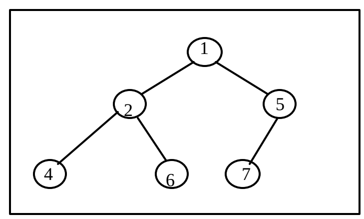



## Build-heap{7,1,5,4,2,6}






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





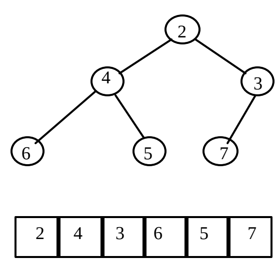
## Running Time

The leaves are at height 0. Consider the nodes at height i.

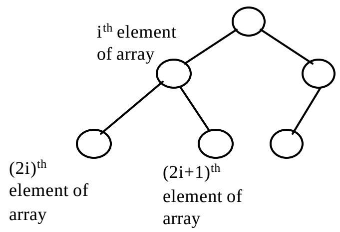
- How long does it take to fix a subtree rooted at height i assuming its children root heaps?
- How many nodes are at height i?
- What is the running time of Build-heap?



## Implementing a heap in an array



## Array Indexing






## Heap-sort(S)

H=Build-heap(S) → O(n)
For i=1 to n
  S(i) = Extract-min(H) → O(lg n)
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

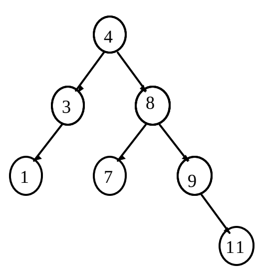


## Binary Search Tree for S

- 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 $\leq$ X.key
  - If node Y in the right subtree of X thenY.key $\geq$ X.key



## BST



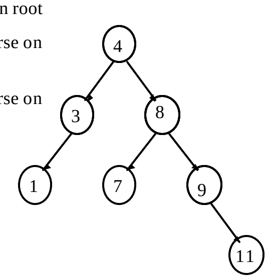



## BST – Find (x)

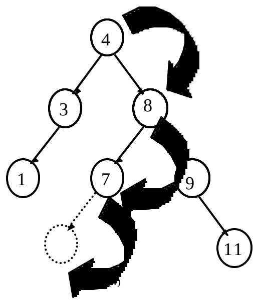
If root.key=x return root

If x<root.key recurse on left subtree

If x>root.key recurse on right subtree

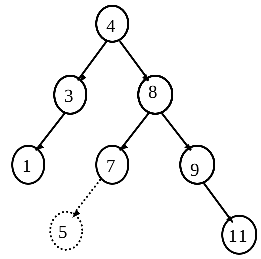


## Insert (5)



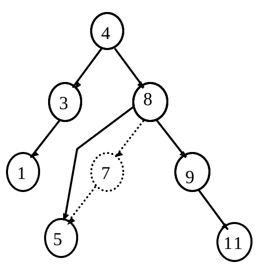



## Delete(5)
## (Leaf is easy)



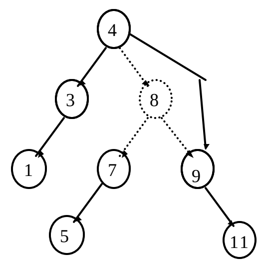



## Delete(7)
## (Node with 1 child is easy)



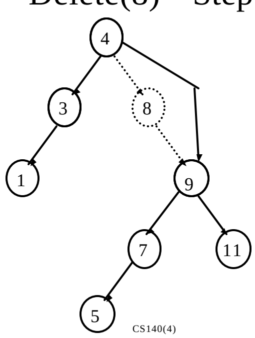



## Delete(8) - Step 1






## Delete(8) – Step 2






## Operation Run Time

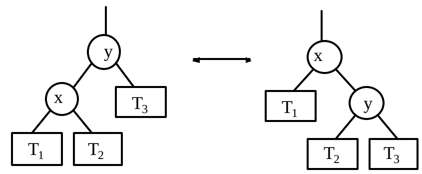
- Search(x) – O(h)
- Insert(x) – O(h)
- Delete(x) – O(h)

## Keeping a good balance …

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



## Rotations






## 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?**



## 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

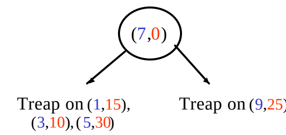
T=(1,15), (3,10), (5,30), (7,0), (9,25)

- Root:
- Left subtree:
- Right subtree:



## Treap

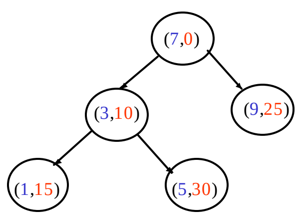
T=(1,15), (3,10), (5,30), (7,0), (9,25)






## Treap

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






## Treaps

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

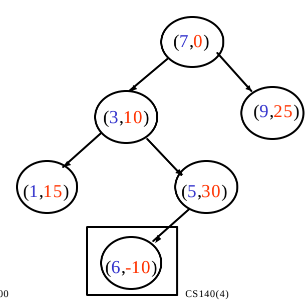


## Binary Tree Operations

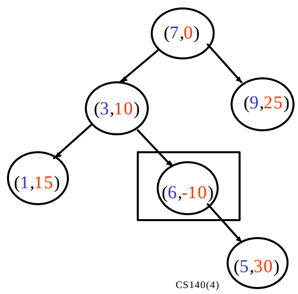
- Insert
- Delete (homework)



## Treap: Insert (6,-10)

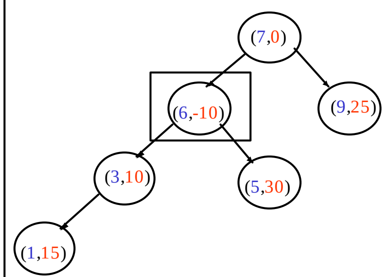


## Treap: Rotate



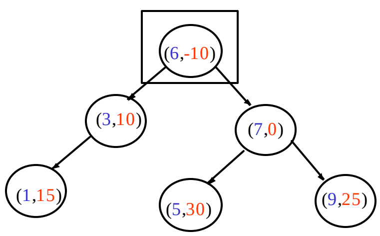



## Treap: Rotate






## Treap: Rotate






## 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(lg(n)).