Heaps

https://cs.pomona.edu/classes/cs140/
Notes

• Grades on gradescope and Sakai

• Make sure you check my grading (regrade requests)

• Checkpoint 3 coming up
  • Connectivity
  • BFS and DFS
  • Topological Orderings
  • Graph Representations
  • Kosaraju's Algorithm (and SCCs)
  • Dijkstra's Algorithm (and SSSP)
Outline

Topics and Learning Objectives
• Discuss data structure operations
• Cover heap sort
• Discuss heaps

Exercise
• Heap practice
Extra Resources

• Introduction to Algorithms, 3rd, chapter 6
Data Structures

Used in essentially every single programming task that you can think of

• What are some examples of data structures?
List some data structures.
Data Structures

Used in essentially every single programming task that you can think of

• What are some examples of data structures?
• What are some example programs?

What do they do?

• They organize data so that it can be effectively accessed.

• A data structure is not necessarily a method of laying out data in memory
• It is a way of logically thinking about your data.
The Heap Data Structure (not heap memory)

A container for objects that have key values
(Sometimes called a “Priority Queue”)

Operations:

- Insertion : $O(\lg n)$
- Extract-min (or max) : $O(\lg n)$
- Heapify : $O(n)$ for batched insertions
- Arbitrary Deletion : $O(\lg n)$

- Good for continually getting a minimum (or maximum) value
Heap used to improve algorithm

Selection sort

• Continually look for the smallest element
• The element currently being considered is in blue
• The current smallest element is in red
• Sorted elements are in yellow
Heap used to improve algorithm

Selection sort
• Continually look for the smallest element
• The element currently being considered is in blue
• The current smallest element is in red
• Sorted elements are in yellow
Heap used to improve algorithm

Selection sort
• Continually look for the smallest element
• The element currently being considered is in blue
• The current smallest element is in red
• Sorted elements are in yellow
Heap used to improve algorithm

Selection sort
• Continually look for the smallest element

What is the runtime of selection sort?
How can we make it faster with a heap? (Breakout)

\[ O(n) + n \cdot O(\log n) = O(n \log n) \]

With a heap: \( O(n^2) \rightarrow O(n \log n) \)
• Insert all elements into a heap: \( n \)
• Extract each element: \( n \cdot \log n \)
Example: Event Manager

Uses a priority queue (synonym for Heap)

Example: simulation or game

- play sounds
- render animation
- detect collisions
- register input
Heap Implementation

Conceptually you should think of a Heap as a binary tree.
It is actually implemented using an array (why?)

Heap Property: for any given node \( x \),

1. \( \text{key}[x] \leq \text{key}[x\text{’s left child}] \), and
2. \( \text{key}[x] \leq \text{key}[x\text{’s right child}] \)

Where is the minimum key?

Root

Locality

- No pointer following
- Fewer heap-allocations
- No pointer storing
Heap Implementation

Note: Heaps are not unique

You can have multiple different configurations that hold the same data
How do you calculate the index of a node’s parent?

\[ \text{parent\_index} = \left( \text{node\_index} - 1 \right) \div 2 \]
How do you calculate the indices of a node’s children?

\[
\text{parent\_index} = \frac{\text{node\_index} - 1}{2}
\]

\[
\text{left\_child\_index} = 2 \times \text{node\_index} + 1
\]

\[
\text{right\_child\_index} = 2 \times (\text{node\_index} + 1)
\]
Exercise
Insert: 7

Where should it go?
Insert: 7

Where should it go?

What can't we have gaps in the array?
Insert: 7
Insert: 10
Insert: 10
Insert: 5
Insert: 5

\[ \text{parent_index} = \left( \text{node_index} - 1 \right) \div 2 \]
Insert: 5

parent_index = (node_index - 1) \[\div\] 2
Insert: 5

parent_index = (node_index - 1) // 2
Insert: 5

parent_index = \( (\text{node_index} - 1) \div 2 \)
What is the running time of an insertion?

This is sometimes called “Bubbling-Up”
What node do we put at the root in place of 4?
What node do we put at the root in place of 4?

What if we choose a child?
Extract Min
Extract Min
Extract Min
What node do we put in place of 4?

We are guaranteed to not leave a gap if we choose the last node.
Extract Min

left_child_index = 2 * node_index + 1

left_child_index = 2 * (node_index + 1)
Do we swap with the 4 or the 5?

Extract Min

```
left_child_index = 2 * node_index + 1

left_child_index = 2 * (node_index + 1)
```
Extract Min

left_child_index = 2 * node_index + 1
left_child_index = 2 * (node_index + 1)
Extract Min

left_child_index = 2 * node_index + 1

left_child_index = 2 * (node_index + 1)
Extract Min

left_child_index = 2 \times \text{node_index} + 1

left_child_index = 2 \times (\text{node_index} + 1)
left_child_index = 2 * node_index + 1
left_child_index = 2 * (node_index + 1)
Extract Min

left_child_index = 2 * node_index + 1

left_child_index = 2 * (node_index + 1)
FUNCTION Dijkstra(G, start_vertex)

found = {}
lengths = {v: INFINITY FOR v IN G.vertices}

found.add(start_vertex)
lengths[start_vertex] = 0

WHILE found.length != G.vertices.length
    FOR v IN found
        FOR vOther, weight IN G.edges[v]
            IF vOther NOT IN found
                vOther_length = lengths[v] + weight
                IF vOther_length < min_length
                    min_length = vOther_length
                    vMin = vOther
        found.add(vMin)
        lengths[vMin] = min_length

RETURN lengths

How many times does the outer loop run?
O(n)

How many times do the inner two loops run?
O(m)
FUNCTION Dijkstra(G, start_vertex)

found = {}
lenghts = {v: INFINITY FOR v IN G.vertices}

found.add(start_vertex)
lenghts[start_vertex] = 0

WHILE found.length != G.vertices.length
  FOR v IN found
    FOR vOther, weight IN G.edges[v]
      IF vOther NOT IN found
        vOther_length = lengths[v] + weight
        IF vOther_length < min_length
          min_length = vOther_length
          vMin = vOther
      found.add(vMin)
lenghts[vMin] = min_length

RETURN lengths
FUNCTION Dijkstra(G, start_vertex)
    found = {}
    lengths = {v: INFINITY FOR v IN G.vertices}

    found.add(start_vertex)
    lengths[start_vertex] = 0

    WHILE found.length != G.vertices.length
        FOR v IN found
            FOR vOther, weight IN G.edges[v]
                IF vOther NOT IN found
                    vOther_length = lengths[v] + weight
                    IF vOther_length < min_length
                        min_length = vOther_length
                        vMin = vOther
            found.add(vMin)
            lengths[vMin] = min_length

    RETURN lengths

What is the running time?

Store vertices in heap

\[ \text{\(O(m \lg n)\)} \]

State of the art of Dijkstra’s:
\[ \text{\(O(m + n \lg n)\)} \]
(uses Fibonacci heap)