Heaps

https://cs.pomona.edu/classes/cs140/
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?
• 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)

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
But it is 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?

- 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?

```
parent_index = (node_index - 1) // 2
```
How do you calculate the indices of a node’s children?

\[
\begin{align*}
\text{parent\_index} &= (\text{node\_index} - 1) \div 2 \\
\text{left\_child\_index} &= 2 \times \text{node\_index} + 1 \\
\text{right\_child\_index} &= 2 \times (\text{node\_index} + 1)
\end{align*}
\]
Exercise
Insert: 7

Where should it go?
Insert: 7

Where should it go?

We don’t want gaps in the array. They signal the end of a search.
Insert: 7
Insert: 10
Insert: 10
Insert: 5
parent_index = (node_index - 1) // 2

Insert: 5
Insert: 5

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

\[ \text{parent\_index} = (\text{node\_index} - 1) \div 2 \]
Insert: 5

parent_index = (node_index - 1) // 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

4 4 5 9 8 9 11 13 7 10 12

9

11 13 7 10 12

?
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?

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

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

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

left_child_index = 2 * node_index + 1

left_child_index = 2 * (node_index + 1)
Extract Min

left_child_index = 2 \times node_index + 1

left_child_index = 2 \times (node_index + 1)
Extract Min

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

Bubbling-Down
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?

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 = {}
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?

We can bring this down to O(m lg m) with a simple change.

State of the art of Dijkstra’s: O(m + n lg n) (uses Fibonacci heap)
def dijkstras_heap(adjacency_list, start_vertex):
    """Dijkstra's Algorithm implemented with all vertices placed in a heap.
    This version of Dijkstra's Algorithm has a running time of O(m lg m).
    """

    n = len(adjacency_list)

    path_lengths = {v: inf for v in adjacency_list}
    predecessors = {v: None for v in adjacency_list}

    path_lengths[start_vertex] = 0
    predecessors[start_vertex] = None

    found = set()
    vertex_min_heap = [(path_lengths[start_vertex], start_vertex)]

    while len(found) != n:
        vfrom_length, vfrom = heappop(vertex_min_heap)
        found.add(vfrom)

        for vto, weight in adjacency_list[vfrom]:
            path_length = vfrom_length + weight
            if path_length < path_lengths[vto]:
                path_lengths[vto] = path_length
                predecessors[vto] = vfrom

                heappush(vertex_min_heap, (path_lengths[vto], vto))

    return path_lengths, predecessors
while len(found) != n:
    vfrom_length, vfrom = heappop(vertex_min_heap)
    found.add(vfrom)
    for vto, weight in adjacency_list[vfrom]:
        path_length = vfrom_length + weight
        if path_length < path_lengths[vto]:
            path_lengths[vto] = path_length
            predecessors[vto] = vfrom
            heappush(vertex_min_heap, (path_lengths[vto], vto))
def print_path(end_vertex, predecessors):
    path = [end_vertex]
    pred = predecessors[end_vertex]

    while pred is not None:
        path.append(pred)
        pred = predecessors[pred]

    print(" -> ".join([str(v) for v in reversed(path)]))
Dijkstra’s Algorithm Correctness

**Theorem:**
• Dijkstra’s algorithm will find the shortest path from the start vertex to every other vertex on any graph with non-negative weights.

**Proof using a loop invariant.** **Loop predicate:**
• At the start of each iteration of the while loop, the shortest path has been found for every vertex in the found set
Initialization

• Initially, the found set is empty. So, the invariant is trivially true.

Loop predicate/invariant: At the start of each iteration of the while loop, the shortest path has been found for every vertex in the found set.

```python
...
found = set()
...
while len(found) != n:
    vfrom_length, vfrom = heappop(vertex_min_heap)
    found.add(vfrom)

    for vto, weight in adjacency_list[vfrom]:
        path_length = vfrom_length + weight
        if path_length < path_lengths[vto]:
            path_lengths[vto] = path_length
            predecessors[vto] = vfrom

            heappush(vertex_min_heap, (path_lengths[vto], vto))
```
Maintenance (1)

- Assume all previous iterations have produced the correct shortest path for all vertices in the found set.
- For purposes of a contradiction, assume that when a vertex \( u \) is added to the found set its path length is \textbf{not} optimal.
- At the time \( u \) is found we must have some path to \( u \)

**Loop predicate/invariant:** At the start of each iteration of the while loop, the shortest path has been found for every vertex in the found set.

```python
while len(found) != n:
    vfrom_length, vfrom = heappop(vertex_min_heap)
    found.add(vfrom)
    for vto, weight in adjacency_list[vfrom]:
        path_length = vfrom_length + weight
        if path_length < path_lengths[vto]:
            path_lengths[vto] = path_length
            predecessors[vto] = vfrom
            heappush(vertex_min_heap, (path_lengths[vto], vto))
```
• Assume all previous iterations have produced the correct shortest path for all vertices in the found set.

• For purposes of a contradiction, assume that when a vertex $u$ is added to the found set its path length is \textbf{not} optimal.

• At the time $u$ is found we must have some path to $u$

Loop predicate/invariant: At the start of each iteration of the while loop, the shortest path has been found for every vertex in the found set.

While $\text{len(found)} \neq n$:

\begin{verbatim}
    vfrom_length, vfrom = heappop(vertex_min_heap)
    found.add(vfrom)
    for vto, weight in adjacency_list[vfrom]:
        path_length = vfrom_length + weight
        if path_length < path_lengths[vto]:
            path_lengths[vto] = path_length
            predecessors[vto] = vfrom
            heappush(vertex_min_heap, (path_lengths[vto], vto))
\end{verbatim}
Maintenance (1)

• For purposes of a contradiction, assume that when a vertex $u$ is added to the found set its path length is **not** optimal.

• At the time $u$ is found we must have some path to $u$.

• To have a shorter path to $u$, it must go through some vertex $k$ not in found.

• But since we only have positive edges, a shorter path going through $k$, means that $k$ must have been chosen before $u$. **Contradiction.**

---

**Loop predicate/invariant:** At the start of each iteration of the while loop, the shortest path has been found for every vertex in the found set.

```python
while len(found) != n:
    vfrom_length, vfrom = heappop(vertex_min_heap)
    found.add(vfrom)

    for vto, weight in adjacency_list[vfrom]:
        path_length = vfrom_length + weight
        if path_length < path_lengths[vto]:
            path_lengths[vto] = path_length
            predecessors[vto] = vfrom

    heappush(vertex_min_heap, (path_lengths[vto], vto))
```
Termination

• The loop terminates when all vertices have been added to the found set.

• Given the loop invariant the shortest path to all vertices have been calculated.

Loop predicate/invariant: At the start of each iteration of the while loop, the shortest path has been found for every vertex in the found set.

while len(found) != n:
    vfrom_length, vfrom = heappop(vertex_min_heap)
    found.add(vfrom)
    for vto, weight in adjacency_list[vfrom]:
        path_length = vfrom_length + weight
        if path_length < path_lengths[vto]:
            path_lengths[vto] = path_length
            predecessors[vto] = vfrom
            heappush(vertex_min_heap, (path_lengths[vto], vto))