Heaps-PriorityQueues

August 7, 2020

1 Priority Queues and Heap

- https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Heaps.html

1.1 Table of Contents

1.2 Priority Queues

- in real-life and in computing applications, we may have to choose the next "most important" from a collection of people, tasks, or objects
  - doctors in a hospital emergency room often choose to see next "most critical" patient
  - operating systems picks programs (jobs) with the highest priority
- when collection of objects is organized by importance or priority, we call this a **priority queue**
  - normal queue is not efficient as it takes $\Theta(n)$ time to search for the next highest priority element

1.3 Priority Queue Applications

- can be applied to solving graph problems such as single-source shortest paths (SSSP) and minimal-cost spanning tree (MST)

1.4 Representing Priority Queue

- how should we effectively represent priority queue?
- a list whether sorted or not, will require $\Theta(n)$ time for either insertion or removal
- BST would require $\Theta(nlogn)$ time in the average case; however BST can become unbalanced leading to bad performance $O(n^2)$
- Heap data structrue is an efficient way!

1.5 Heap

- **heap** data structure is used to represent priority queues
- heap is also the name for a memory segment (**free store**)

- two properties:
  1. it is a complete binary tree
     - heaps are nearly always implemented using the array representation
  2. the values stored in a heap data structure are partially ordered
     - there's a relationship between the value stored at any node and the values of its children
     - no relationship between the siblings (unlike BST)
- two types of heap:
  1. max heap
     - every node stores a value that is **greater** than or equal to the value of either of its children
     - by its definition, root stores the maximum of all values in the tree
  2. min heap
     - every node stores a value that is **less** than or equal to that of its children
     - by its definition, root stores the minimum of all values in the tree
- Heapsort uses max heap
- Replacement Selection algorithm used for external sorting uses min heap

1.6 Building Heap

- Two ways:
  1. push heap one element at a time
  2. make heap from given list of elements

1.7 Push Heap

- similar to: https://en.cppreference.com/w/cpp/algorithm/push_heap
- useful when all elements are not available at once
- algorithm steps:
  1. first copy the data, V at the last index
  - move V to the right place by comparing to its parent's value
    1. if the value of V is less than or equal to the value of its parent, it is in the correct position
    * if the value of V is greater than that of its parent, the two elements swap positions
    * repeat 2 until V reaches its correct position
- visualize heap push/insert: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Heaps.html
- running time complexity:
  - since the height of a complete binary tree with n nodes is $(logn)$, each call to push takes $O(logn)$ time in the worst case, (move from the bottom to the top)
  - so, takes $O(nlogn)$ time in the worst case

1.7.1 push heap operation

- let's say we have values from 1..7 we want to push to a max heap one element at a time.
- final heap looks like this:
- Heap in above figure is built by pushing one element at a time with a total of (11 swaps):
  - (2, 1), (3, 2), (4, 1), (4, 2), (4, 3), (5, 3), (5, 4), (6, 2), (6, 5), (7, 5), (7, 6)
- visualize it here pushing one element at a time: https://visualgo.net/en/heap

1.8 Make Heap

- similar to: https://en.cppreference.com/w/cpp/algorithm/make_heap
- useful when all n values are available at the beginning of the building process
- make heap is faster than push heap one element at a time

1.8.1 make heap operation

- let's say we have values 1..7 already stored in some sequence data structure like vector as shown in the following figure:
- start pushing down from 2nd last level and up
- with the total of 4 swaps (3, 7), (2, 5), (1, 7), (1, 6)
  - the final max heap looks like the following:
- visualize make heap operation: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Heaps.html#build-a-heap

1.8.2 algorithm steps (based on induction/recursion)

1. suppose that left and right subtrees of the root are already heaps, and R is the name of the element at the root
2. two possibilities:
   1. R has value greater than or equal to both children (done!)
   2. R has a value less than one or both of its children
      - **"push down"** R until it's greater than its children, or is a leaf node
      - keep exchanging R with the child that has greater value resulting heap

1.8.3 Running time complexity

- make heap takes $O(n)$ in the worst case better than $O(nlogn)$ (building heap one element at a time)
- Compared to BST:
  - better than $O(n^2)$ worst-case and $O(nlogn)$ average-case time required to build BST

1.9 Pop Heap

- remove and return the maximum value from the max heap
- similar to: https://en.cppreference.com/w/cpp/algorithm/pop_heap
- algorithm steps:
  1. swap the first and the last positions
  2. decrement the heap size by one
  3. since it's no longer a max heap, push the top element down as appropriate
  4. return the max element
- visualize it here: Removing from the heap section- https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Heaps.html
- because the heap is $logn$ levels deep, the cost of deleting the maximum element is $\Theta(logn)$ in the average and worst cases

1.10 Max Heap Implementation

- implemented using array-based complete binary tree

- https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Heaps.html

[1]:
```
#include <iostream>
#include <string>
#include <vector>
#include <cassert>
#include <sstream>
#include <algorithm>

using namespace std;
```

[2]:
```
// Max-heap ADT
// Max Priority-Queue
template<class T>
class MaxHeap {
    private:
        vector<T> heap;
        size_t max_size;
        size_t size;

        //return true if given pos is a leaf position, false otherwise
        bool isLeaf(size_t pos) {
            return (pos >= size/2 && (pos < size));
        }

        // return leftChild's index given a parent's index
        int leftChild(size_t parentIndex) {
            if (parentIndex >= size/2) return -1;
            return 2*parentIndex + 1;
        }

        // return rigthChild's index given a parent's index
        int rightChild(size_t parentIndex) {
            if (parentIndex >= (size-1)/2) return -1;
            return 2*parentIndex + 2;
        }

        // return parent's index given child's index
        int parent(size_t childIndex) {
            if (childIndex <= 0) return -1;
            return (childIndex-1)/2;
        }

        // heapify contents of heap
        // https://en.cppreference.com/w/cpp/algorithm/make_heap
        void makeHeap() {
            // start pushing down from 2nd last level and up
```

```
            for(int i=size/2 - 1; i>=0; i--) pushDown(i);
        }

        // push the element down to its correct place
        void pushDown(size_t pos) {
            if ((pos < 0) || (pos >= size)) return; //illegal position
            // push down until the pos is a leaf or heap is built
            while(!isLeaf(pos)) {
                // find the index of the larger of the two children
                int j = leftChild(pos); // let's say left child is greater
                // check if the right sibling is greater than left
                // first check if j+1 doesn't go outof bounds or right child exists
                if (( j < (size-1)) && (heap[j] < heap[j+1]))
                    ++j; // j+1 is index of child with greater value
                // if value at pos is larger than its larger of the two children; its a heap
                if (heap[pos] >= heap[j]) return;
                swap(heap[pos], heap[j]); //
                pos = j; //move down to new pos
            }
        }

    public:
        MaxHeap(size_t max_size, vector<T> items) {
            this->max_size = max_size;
            this->heap.resize(max_size);
            this->size = items.size();
            heap = items;
            makeHeap();
        }

        //return the current size of the heap
        size_t heapSize() const { return this->size; }

        bool isEmpty() const { return this->size == 0; }

        bool isFull() const { return this->size == this->max_size; }

        // insert a value into heap
        // https://en.cppreference.com/w/cpp/algorithm/push_heap
        // insert value at the end and shift up to its correct location
        void push(const T &value) {
            if (isFull()) { // Heap is full...
                return;
            }
            int curr = size++; // use size as current index and increment it
```

```
            heap[curr] = value; // start at the end
            // now shift up until curr's parent's key > curr's key
            while ((curr > 0) && (heap[parent(curr)] < heap[curr])) {
                swap(heap[parent(curr)], heap[curr]);
                curr = parent(curr);
            }
        }

        // remove and return the max value from the heap
        // https://en.cppreference.com/w/cpp/algorithm/pop_heap
        T pop() {
            assert(size > 0); // can't pop from empty heap
            swap(heap[0], heap[--size]); //swap maximum with last value
            if (size != 0) // not on last element
                pushDown(0); //put new heap root val in correct place
            return heap[size];
        }
};
```

[3]:
```
// Test Max-heap
// see building the heap visualization in the above open-dsa link
vector<int> nums = {1, 2, 3, 4, 5, 6, 7};
```

[4]:
```
// make heap from nums vector
MaxHeap<int> heap(100, nums);
```

[5]:
```
cout << "heap size = " << heap.heapSize() << endl;
// pop max element
cout << "max value = " << heap.pop() << endl;
```

```
heap size = 7
max value = 7
```

[6]:
```
// push an element
heap.push(8);
cout << "heap size = " << heap.heapSize() << endl;
```

```
heap size = 7
```

[7]:
```
cout << "max value = " << heap.pop() << endl;
```

```
max value = 8
```

[8]:
```
while (!heap.isEmpty()) {
    cout << heap.pop() << endl;
}
```

```
6
```

```
5
4
3
2
1
```

[9]:
```
cout << "heap size = " << heap.heapSize() << endl;
```

```
heap size = 0
```

1.10.1 Storing jobs in PriorityQueue

[10]:
```
class Job {
    public:
        int ID;
        int priority;
        string name;
        bool operator>=(const Job& other) {
            return this->priority >= other.priority;
        }

        bool operator<(const Job& other) {
            return this->priority < other.priority;
        }

        void print() {
            cout << "ID: " << this->ID << " Priority: " << this->priority << "␣
 ↪Name: " << this->name << endl;
        }
};
```

[11]:
```
vector<Job> jobs = {{1, 10, "Print"}, {3, 30, "Write"}, {2, 20, "Read"}};
```

[12]:
```
MaxHeap<Job> jobsQueue(50, jobs);
```

[13]:
```
cout << "heap size = " << jobsQueue.heapSize() << endl;
```

```
heap size = 3
```

[14]:
```
Job j;
```

[15]:
```
j = jobsQueue.pop();
j.print();
```

```
ID: 3 Priority: 30 Name: Write
```

[16]:
```
jobsQueue.push({5, 100, "Connect"});
```

[17]:
```
cout << "queue size = " << jobsQueue.heapSize() << endl;
j = jobsQueue.pop();
cout << "highest priority job = ";
j.print();
```

```
queue size = 3
highest priority job = ID: 5 Priority: 100 Name: Connect
```

1.11 Using MaxHeap as MinHeap

- reverse the ordering of cost/weight value that determines priority
- OR
  - -negate the values of cost/weight that determine priority

[18]:
```
// Reverse the ordering of values to create Min Priority Queue/Min Heap
class Task {
    public:
        int ID;
        int cost; // priority set based on cost; lower the cost; higher the␣
↪priority!
        bool operator>=(const Task& other) {
            return this->cost <= other.cost;
        }

        bool operator<(const Task& other) {
            return this->cost > other.cost;
        }

        void print() {
            cout << "ID: " << this->ID << " cost: " << this->cost << endl;
        }
};
```

[19]:
```
vector<Task> tasks = {{10, 200}, {1, 10}, {5, 50}};
```

[20]:
```
MaxHeap<Task> minPq(50, tasks);
```

[21]:
```
// process all the tasks based on cost; with smaller cost first
while (!minPq.isEmpty()) {
    Task t = minPq.pop();
    t.print();
}
```

```
ID: 1 cost: 10
ID: 5 cost: 50
ID: 10 cost: 200
```

[22]:
```
// MinHeap: negate the values of weights/costs
// use the same values as we used in the first example above;
vector<int> nums1 = {1, 2, 3, 4, 5, 6, 7};
```

[27]:
```
// negate each value
for_each(nums1.begin(), nums1.end(), [](int &n){ n *= -1; });
```

[28]:
```
for (auto &n: nums1) { cout << n << " "; }
```

```
-1 -2 -3 -4 -5 -6 -7
```

[29]:
```
// build min-heap
MaxHeap<int> minPqInts(10, nums1);
```

[31]:
```
// process heap/MinPQ one element at a time
while(!minPqInts.isEmpty()) {
    cout << minPqInts.pop() << endl;
}
```

```
-1
-2
-3
-4
-5
-6
-7
```

1.12 Kahoot.it

- https://play.kahoot.it/v2/lobby?quizId=a1f75bcb-ccfa-4d8a-b777-bcd71f7de2fc

1.13 Exercises

1. Consider a node R of a complete binary tree whose value is stored in position i of an array representation for the tree. If R has a parent, where will the parent's position be in the array?
    1. $2 * i + 1$
    2. $i + 1$
    - $\left\lfloor \frac{i-1}{2} \right\rfloor$
    - $2 * i + 2$

- Which of these is true statement about the worst-case time for operations on heaps?
    1. Neither insertion nor removal are better than linear
    - Insertion is better than linear, but removal is not
    - Removal is better than linear, but insertion is not
    - Both insertion and removal are better than linear
- In a max-heap containing n elements, what is the position of the element with the max value?
    1. $n + 1$
    - 0
    - Possibly in any leaf nodes

- $2 * n + 1$
- n
- $n - 1$
- $2 * n + 2$

- In a max-heap containing n elements, what is the position of the element with the least/min value?
  1. $n + 1$
  - 0
  - Possibly in any leaf node
  - $2 * n + 1$
  - n
  - $n - 1$
  - $2 * n + 2$
- In a min-heap containing n elements, what is the position of the element with the least/min value?
  1. $n + 1$
  - 0
  - Possibly in any leaf node
  - $2 * n + 1$
  - n
  - $n - 1$
  - $2 * n + 2$
- In a min-heap containing n elements, what is the position of the element with the max value?
  1. $n + 1$
  - 0
  - Possibly in any leaf node
  - $2 * n + 1$
  - n
  - $n - 1$
  - $2 * n + 2$

[ ]:

```

```