Heap

Edwin Ahlstrand

Fall 2022

Introduction

This assignment covers a data structure called heap, which functions as a priority queue where elements are assigned a priority and the one with the highest priority is first served. The heap is implemented in three different ways, as a list, a tree and an array. Finally they are all benchmarked and compared.

Priority queues / heaps

The heaps consist of two methods, one add and one remove. The first method takes creates a node with a given priority to add to the queue, and the last one removes the node with the highest priority in the list and returns it's value.

List

The heap using a list can be implemented in two ways, the first way is with an add function with time complexity O(1) and remove with O(n), the second version is vice versa.

With the first version, the add function simply adds the item at the top of the queue without regard to its priority, and the remove function searches the list for the highest priority and removes it.

Meanwhile with the second version, the add function adds the item in its correct position with highest priority at the top, and the remove function simply removes the highest item.

These two variants will be compared later under the benchmark section.

Array

The array implementation of a heap works in a very smart way. The value at index 0 represents the root, and a parent element has it's children at positions $2 * n + 1$ for the left and $2 * n + 2$ for the right, where n is the index of the parent. Therefore when adding elements to the array, they are added to the first empty position (after the last used position) and then moves upwards through it's parents if it is smaller. In this way, it is easy to keep the tree balanced since there should be no empty gaps between the used indexes.

When removing elements, it is done by returning the root (which is highest priority) and then selecting the last element as the new root and letting it sink towards it's right position.

The code for adding and removing elements is following:

```java
@Override public int add(int value){
    if(last_pos<0){  // Empty
        heap[0] = value;
        last_pos = 0;
        return 0;
    }
    heap[++last_pos] = value;  // Add last
    int current_pos = last_pos;  // Start at last
    int current_value = value;  // Value of last
    int parent_pos = current_pos%2 == 0 ? (current_pos-2)/2 : (current_pos-1)/2;
    int parent_value = heap[parent_pos];  // Parent value
    while(current_value < parent_value){  // If value needs to move up
        heap[parent_pos] = current_value;  // Switch values (1)
        heap[current_pos] = parent_value;  // Switch values (2)
        current_pos = parent_pos;  // Update new position
        if (current_pos == 0) // If at top
            return 0;
        current_value = heap[current_pos];  // Update current value
        parent_pos = current_pos%2==0 ? (current_pos-2)/2 : (current_pos-1)/2;
        parent_value = heap[parent_pos];  // Update parent value
    }
    return 0;
}
```

```
@Override public Integer remove(){
    if(last_pos<0)  // Empty, do nothing
        return null;
    Integer value = heap[0];  // Value to be returned
    heap[0] = heap[last_pos];  // Select root as last
    int current_pos = 0;
    int left_pos = current_pos*2+1;
    int right_pos = current_pos*2+2;
    while(left_pos <= last_pos && right_pos <= last_pos){  // Move down
        if(heap[current_pos] <= heap[left_pos]
        && heap[current_pos] <= heap[right_pos])  // If in right position
            break;
        else if(heap[left_pos] <= heap[right_pos]){  // Swap with left child
            Integer tmp = heap[current_pos];
            heap[current_pos] = heap[left_pos];
            heap[left_pos] = tmp;
            current_pos = left_pos;
        } else if(heap[left_pos] > heap[right_pos]){  // Swap with right child
            Integer tmp = heap[current_pos];
            heap[current_pos] = heap[right_pos];
            heap[right_pos] = tmp;
            current_pos = right_pos;
        }
        left_pos = current_pos*2+1;  // New left position
        right_pos = current_pos*2+2;  // New right position
    }
    heap[last_pos] = null;  // Remove previously last element
    last_pos--;  // Decrease last position
    return value;  // Returns previous roots value
}
```

Tree

The tree implementation of a heap is done using nodes, which hold a priority value and points to two neighbouring nodes called left and right. At the top of the tree is a node called root which all other nodes are connected to.

Adding elements are done by swapping values with the smaller values from the children until a leaf is reached, where a new node with the current value is inserted. A snippet of the code is following:

```
while (true) {  // Go through the tree til leaf
    if (curr_value < curr_node.value) {  // Swaps values
        int tmp = curr_node.value;
        curr_node.value = curr_value;
        curr_value = tmp;
    }else{  // Value larger, go down
        curr_node.size++;
        depth++;
        if (curr_node.left == null) {  // Add to left side
            curr_node.left = new Node(curr_value);
            return depth;
        }else if (curr_node.right == null) {  // Add to right side
            curr_node.right = new Node(curr_value);
            return depth;
        }if (curr_node.left.size < curr_node.right.size) {  // Balance out left sid
            curr_node = curr_node.left;
        }else {  // Balance out right side
            curr_node = curr_node.right;
        }
    }
}
```

Removing a node is done by returning the value of the root and recursively promoting it's children. A snippet of the code is following:

```
public Node removeRecursive(Node curr) {
    curr.size--;  // Lower size of current since will remove
    if (curr.left == null) {  // Left side empty, go right
        curr = curr.right;
        return curr;
    }else if (curr.right == null) {  // Right side empty, go left
        curr = curr.left;
        return curr;
    }else if (curr.left.value > curr.right.value) {  // Promote right
        curr.value = curr.right.value;
        curr.right = removeRecursive(curr.right);
        return curr;
    }else if (curr.left.value < curr.right.value) {  // Promote left
        curr.value = curr.left.value;
        curr.left = removeRecursive(curr.left);
        return curr;
    }
    return curr;
}
```

It is often useful to push a node farther down the heap, this is done using a push method. A snippet from the push method is following:

```
while (true) {
    boolean left_smaller;  // For choosing going left or right
    if(curr.left != null && curr.right != null){  // Left and right not empty
        if (!(curr.left.value < value || curr.right.value < value)) {  // None is s
            return depth;
        }
        left_smaller = curr.left.value<curr.right.value;  // Left or right smaller
    } else left_smaller = curr.left != null;  // If left exist
    depth++;  // Increment depth
    if(left_smaller){  // Go left
        value = curr.value;
        curr.value = curr.left.value;
        curr.left.value = value;
        value = curr.value;
        curr = curr.left;
    }else if (curr.right != null){  // Go right
        value = curr.value;
        curr.value = curr.right.value;
        curr.right.value = value;
        value = curr.value;
        curr = curr.right;
    }else  // Both null
        return depth;
}
```

Some statistics of the depth reached when adding and pushing values in the heap will be presented under the benchmark section.

Benchmarks

Lists and array benchmark

Both the list implementations and the array are compared against each other when adding and removing elements of an increasing amount (n). They are all filled with the same random values in the same order and then removed one by one.

The list implementation with constant add and linear remove functions is called List1, and the one with linear add and constant remove functions is called List2.

The results are the time to finish the add and remove functions, respectively. The benchmark was run 1000 times to get an average time for all of the operations to be completed, which was in turn run 10 times to get the minimum of those.

The benchmarks results are given from the table below with a format as add/remove and size noted as n:

n	**List1**	**List2**	**Array**
100	0.3 / 9.0	5.0 / 0.2	2.5 / 4.5
200	0.6 / 37	18 / 0.4	4.4 / 10
300	0.9 / 82	38 / 0.5	6.6 / 18
400	1.2 / 140	64 / 0.7	9.0 / 24
500	1.5 / 220	100 / 0.9	11 / 32
600	1.8 / 320	140 / 1.1	18 / 62
700	4.7 / 440	200 / 3.8	20 / 72
800	5.4 / 580	260 / 4.4	24 / 90
900	6.0 / 720	320 / 5.0	28 / 100

Table 1: Time to add/remove from different types of heaps. Time is in microseconds.

From the table above, it can be concluded that the array implementation is more consistent, with almost always shorter combined time to add and remove than the two lists. Meanwhile, List2 seems more consistent than List1, but List1 one would still be more preferred in situations where it isn't as important to retrieve the first item quickly.

Tree depth benchmark

The tree implementation is tested with how deep it has to go in layers when adding and pushing a value.

For the add operation, 64 out of 100 unique values from 0-100 are added to the heap and the depth for each add will be saved.

For the push operation, a random number up to 100 is incremented to the same tree and the depth it has to push is saved.

The results show that for the n:th add operation, it has to go to a depth of closest power of two rounded to lowest.

For the push function almost half of the time it doesn't go any deeper, and when it does go deeper it seems evenly split between going $1, 2, 3$ layers, with one time going 4 layers deep.