Theoretical Homework 3

Professor Bauer

Problem 1: Hashing

Insert the following keys one-by-one into an initially empty hash table of size 8. Use the hash function $h(x) = x \mod 8$

$$\{10, 1, 18, 15, 26, 11, 19\}$$

Show the result for

1. A separate chaining hash table (do not rehash if the load factor becomes too large).
2. A table with linear probing, i.e. using the probing function $f(i) = i$.
3. A table with quadratic probing, i.e. using the probing function $f(i) = i^2$.
4. A table that uses double hashing with a secondary hash function, i.e. $f(i) = i * g(x)$, where the secondary hash function is $g(x) = 5 - (x \mod 5)$.

Solution

1. Table with size of 8

0	
1	1
2	$10 \to 18 \to 26$
3	$11 \to 19$
4	
5	
6	
7	15

2. Table with size of 8

0	
1	1
2	10
3	18
4	26
5	11
6	19
7	15

3. Attempting with table with size of 8

0	
1	1
2	10
3	18
4	11
5	
6	26
7	15

Cannot insert 19 so rehash with a table of size 11.

0	11
1	1
2	
3	
4	15
5	26
6	
7	18
8	19
9	
10	10

4. Table with size of 8

0	
1	1
2	10
3	11
4	18
5	19
6	26
7	15

Problem 2

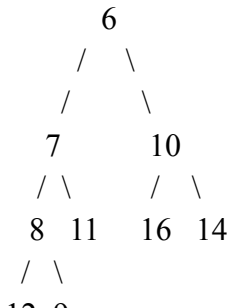
(a) Insert the values 8, 12, 14, 11, 9, 16, 10, 7, 6 into an initially empty binary min heap. Show the heap after each insertion as an array **or** as a tree. You do not need to show each individual percolation step.

(b) Show the **result** of using the linear-time buildHeap algorithm on the same input. You do not have to show individual steps.

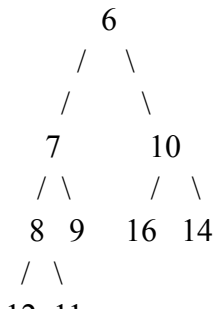
(c) Perform three deleteMin operations on the final heap from part (a). Show the heap after each deleteMin as a tree or array.

Solution

[- , 6, 7, 10, 8, 11, 16, 14, 12, 9]

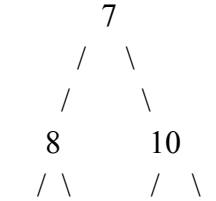


b)

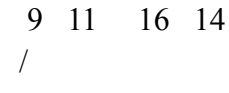


c)

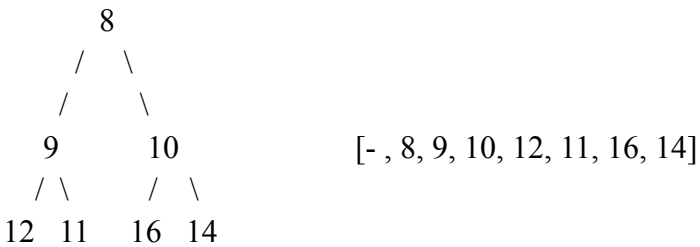
1:



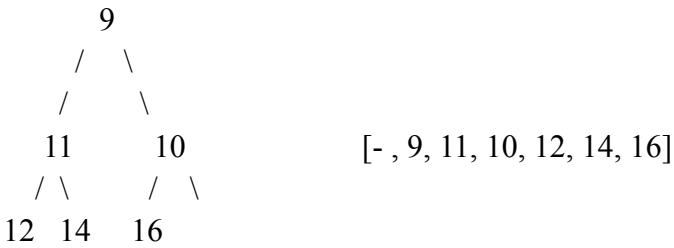
[- , 7, 8, 10, 9, 11, 16, 14, 12]



2:



3:



Problem 3

A min-max heap is a data structure that supports both *deleteMin* and *deleteMax* in O(log N) per operation. The structure is identical to a binary heap, but the heap- order property is that for any node, X, at *even* depth, the element stored at X is smaller than the parent but larger than the grandparent (where this makes sense), and for any node X at *odd* depth, the element stored at X is larger than the parent but smaller than the grandparent.

a) How do we find the minimum and maximum elements?

b) Give an algorithm (in Java-like pseudocode) to insert a new node into the min-max heap. The algorithm should operate on the indices of the heap array.

Solution

(a)

The maximum element is located on the second row (child of the root, the index is 2 or 3) so compare the two elements in the second row to find largest. The minimum element is the root node.

(b)

```
public int insert(new_item){
        heap[++heapsize] = new_item;
        findPosition(heapsize);
}

int findPosition( idx ){
        int depth = calculate heap depth from the index;

        if( idx <= 1;){ //base case
                return
        }

        if (depth is even){
                if (heap[ idx ] > heap[ idx / 2 ]) {
                        swap (idx, idx/2) //swap with parent
                        findPosition( idx /2 )
                 }
                 if (idx/4 != 0 && heap[ idx ] < heap[ idx /4 ]){
                        swap (idx, idx/4) //swap with grandparent
                        findPosition ( idx /4 )
                 }
        }
        else if (it's odd depth){
                 if (heap[ idx ] < heap[ idx / 2 ]) {
                        swap (idx, idx/2) //swap with parent
                        findPosition( idx /2 )
                 }
                 if idx/4 != 0 && (heap[ idx ] > heap[ idx /4 ]){
                        swap (idx, idx/4) //swap with grandparent
                        findPosition ( idx /4 )
                 }
        }
}
```

Problem 4

Instead of a binary heap, we could implement a d-ary heap, which uses d-ary tree. In such a tree, each node has between 0 and d children. As for the binary heap, we assume that a d-ary heap is a *complete* d-ary tree and can be stored in an array. If a node in the d-ary tree is at array position i, where is the parent of this node and where are the children?

Solution

This d-ary tree will be stored in an array. As we build the array, we will place the root at position 0 (in a 0-indexed array). All d children of this root will be placed in array positions 1….d, and the next d^2 children will be from d+ 2 … d^d+d.

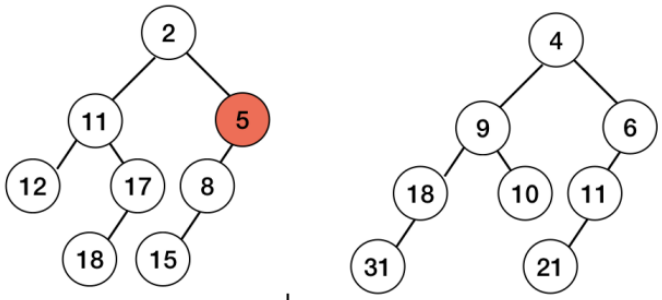
Using this property of building the d-ary tree, we see the parent of a node at position i will be at the position **⌊(i − 1)/d ⌋** , and the children of the node at position i will be at positions **di + 1 through di + d.**

*****In 1-indexed, parent at **i/d**, children at **di through id+d-1 or d(i+1)-1**

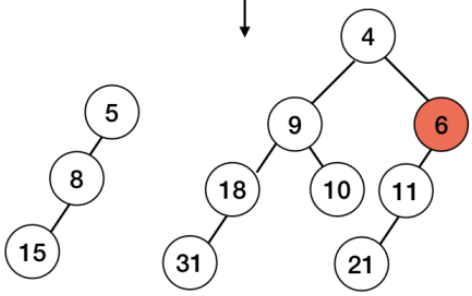
Alternatively, the parent of a node at position i could be located at **⌊(i + (d-2)) / d⌋ or (i-2)/d + 1**, and the children would be between **(i-1)d+2 and di + 1.**

Problem 5 Merge the following two leftist heaps recursively. For each recursive call, show which sub-trees are merged in this step, as well as the merged tree returned by each recursive call.

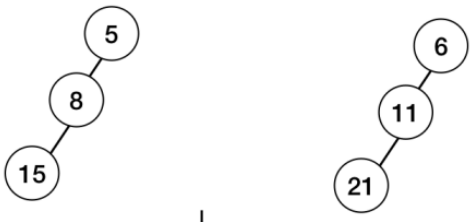
Solution



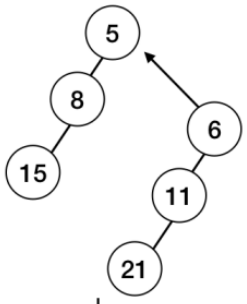
Recursive call



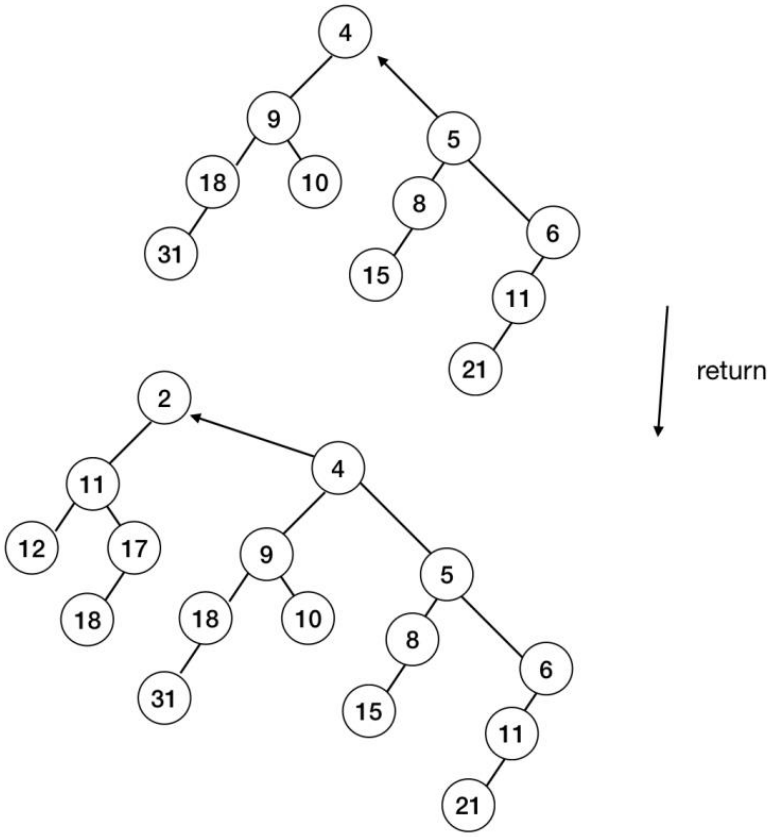
Recursive call



return



return


4
9
5
18
10
8
6
31
15
11
21
return
2
11
4
12
17
9
5
18
18
10
8
6
31
15
11
21