Problem Set 5

Problem 1 Show how to implement a stack ADT using only a priority queue and one additional integer variable.

Solution: Maintain a maxKey variable initialized to 0. On a push operation for element e, call insertItem(maxKey, e) and decrement maxKey. On a pop operation, call removeMinElement and increment maxKey.

Problem 2 Write an algorithm for updating the key of an item in a priority queue, and analyse its time complexity.

Solution: Assume the priority queue is based on a min-heap.

```
Algorithm updateKey(v)
 Input: a node v containing the item
 Output: the heap with the key updated
  {
    // upheap bubbling
    while (v!=NULL && v.key < v.parent.key)
        {
          swap the items of v and its parent;
          v=v.parent;
        }

   // downheap bubbling
   while (v!=NULL && v.key > min{v.left.key, v.right.key})
        {
          let u be the child of v with the smaller key;
         swap the items of v and u;
         v=u;
        }
 }
```

Time complexity analysis: This algorithm performs either upheap bubbling or downheap bubbling. The loop body of each while loop takes O(1) time, and the number of iterations of each while loop is no more than h, where h is the height of the heap. Since h is O(log n), the time complexity of this algorithm is O(log n), where n is the number of items in the heap.

Problem 3 Given a heap T and a key k, give an algorithm to compute all the items in T with keys less than or equal to k. Your algorithm should run in time proportional to the number of items returned.

Solution:

```
Algorithm lessThanOrEqualToKEntries(H, v)
 Input: A heap H and a node v
 Output: A node list L that contains all the entries with keys less than k
  {
   if ( v.key≤k )
     {
       L.add((v.key, v.value));  // add the entry v to the list L
       if (v.leftchild !=null) LessThanOrEqualToKEntries(H, v.leftchild);
       if (v.rightchild !=null) LessThanOrEqualToKEntries(H, v.rightchild);
     }
 }
```

According to the heap order property, there is no node in T storing a key larger than k that has a descendent storing a key less than or equal to k. As a result, this algorithm takes O(n) time, where n is the number of entries returned.

Problem 4 Qantas Airlines wants to give a first-class upgrade coupon to their top log n frequent flyers, based on the number of miles accumulated, where n is the total number of the airlines' frequent flyers. The algorithm they currently use, which runs in O(n log n) time, sorts the flyers by the number of miles flown and then scans the sorted list to pick the top log n flyers. Describe an algorithm that identifies the top log n flyers in O(n) time.

Solution:

```
Algorithm TopKFlyers(A)
 Input: A list A of n flyers
 Output: An array B of the top log n flyers
  {
   Construct a heap H storing all the n flyers, where the key of each flyer Pi is 1/mi
   (mi is the number of miles Pi has flown);
   for (i=0; i< log n; i++)
      B[i]= H.removeMin();
   return B;
  }
```

Running time analysis: It takes O(n) time to construct a heap with n integers as keys by using bottom-up heap construction algorithm. removeMin() takes O(log n) time. Therefore, this algorithm takes $O(n + (\log n)^2)=O(n)$ time.

Problem 5 Suppose two binary trees, T1 and T2, hold entries satisfying the heap-order property, where no entry in each tree exists in the other tree. Describe a method for combining T1 and T2 into a tree T such that T's internal nodes hold the union of the entries in T1 and T2 and T also satisfies the heap-order property. Your algorithm should run in time O(h1+h2) where h1 and h2 are the respective heights of T1 and T2.

Solution:

Algorithm treeUnion(T1, T2)
Input: Two trees T1 and T2 that satisfy the heap-order property.
Output: A tree T that is the union of T1 and T2 and also satisfies the heap-order property.

```
{
  v=T1.removeMin();
  let v be the root of T;
  leftchild(v) = the root of T1;
  rightchild(v) = the root of T2;
  apply the down-heap bubbling to the tree T;
}
```

Running time analysis: T1.removeMin() takes O(h1) time. The down-heap bubbling to the tree T takes O(h2) time as only the down-heap bubbling to the subtree T2 is performed. All other operations take O(1) time. Therefore, this algorithm takes O(h1)+O(h2)+O(1)=O(h1+h2) time.

Problem 6 Give an alternative analysis of the bottom-up heap construction algorithm.

Solution: In the bottom–up heap construction, the number of merge operations is equal to the number of non–leaf nodes. The height of the heap is log n, where n is the total number of nodes. At each level i (i=0, 1, …, log n), the total number of nodes is 2^i. Each node v_k corresponds to one merge operation which takes O(log n – i) time, where log n –i is the height of the subtree rooted at v_k. Therefore, the total time of the heap construction is

$\Sigma_{i=0..\log n} 2^i(\log n - i) = \Sigma_{i=0..\log n} 2^{(\log n - i)} i = 2^{\log n}\Sigma_{i=0..\log n} 2^{-i} i = 2^{\log n}\Sigma_{i=0..\log n} i/2^i$.

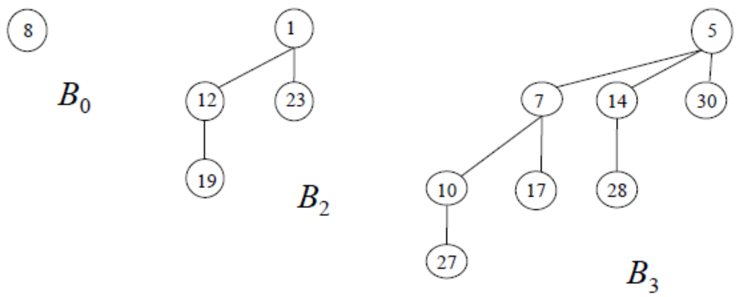
By using induction we can prove that $i \le 2^{i/2}$ holds for i>3. Therefore, we have $\Sigma_{i=0..\log n} i/2^i \le 1+3/8+\Sigma_{i=4..\log n} 1/2^{i/2} = 1+3/8+\Sigma_{i=4..\log n} (1/2^{1/2})^i < 1+3/8+1/(4-2\mathrm{sqrt}(2)) < 2.5$. Hence, $2^{\log n}\Sigma_{i=0..\log n} i/2^i < 2.5*2^{\log n} = 2.5n = O(n)$.

Problem 7 Prove that a binomial tree with 2^n nodes has $\binom{n}{i}$ nodes at depth i ($0 \le i \le n$).

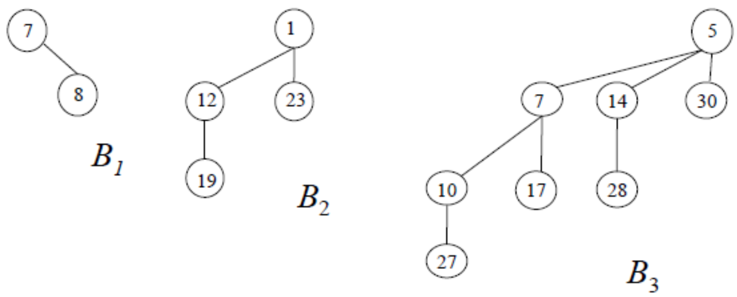
Proof: By the definition B_n, B_n consists of two B_{n-1} with the root with a larger key of one B_{n-1} being the child of the root of the other B_{n-1}. We prove it by induction. For n = 0, only the root of B_n is at depth 0. Therefore, the statement is true. Suppose in B_{n-1}, the number of nodes at depth i is $\binom{n-1}{i}$. Notice that the nodes at depth i − 1 of one B_{n-1} becomes the nodes at depth i for B_n. and the nodes at depth i-1 of the other B_{n-1} remain at the same level. Therefore, in B_n, the number of nodes at depth i is

$$\binom{n-1}{i} + \binom{n-1}{i-1} = \binom{n}{i}.$$

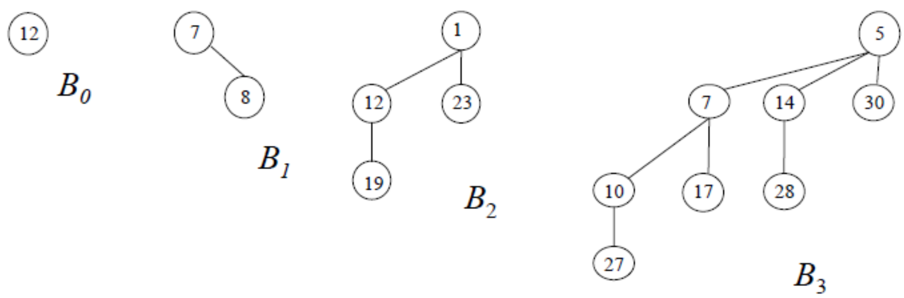
Problem 8 Consider the following binomial heap. Draw the resulting binomial heaps after inserting the keys 7, 12, 20, 24, 25 and 25, respectively.



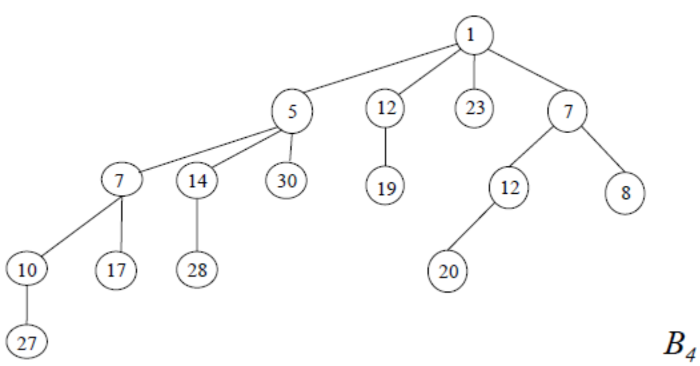
Solution: After insert 7:



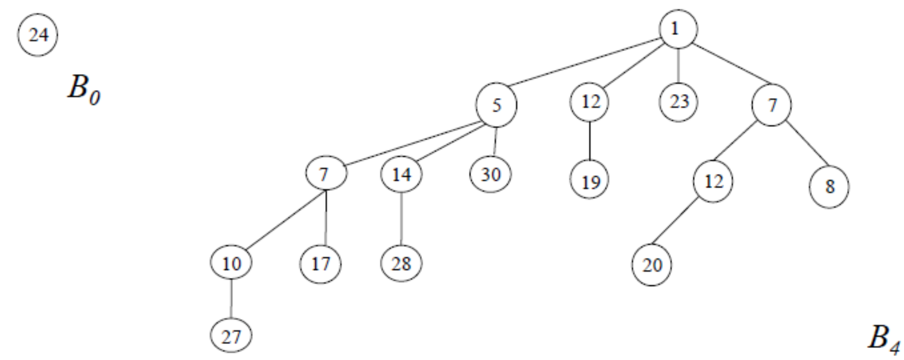
After Insert 12:



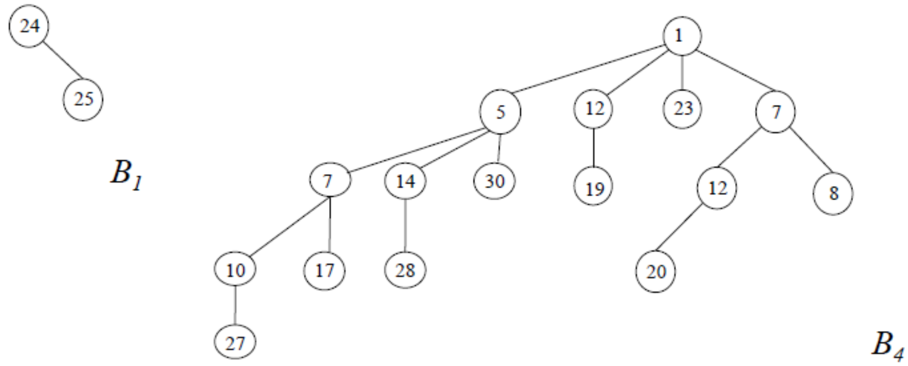
After insert 20:



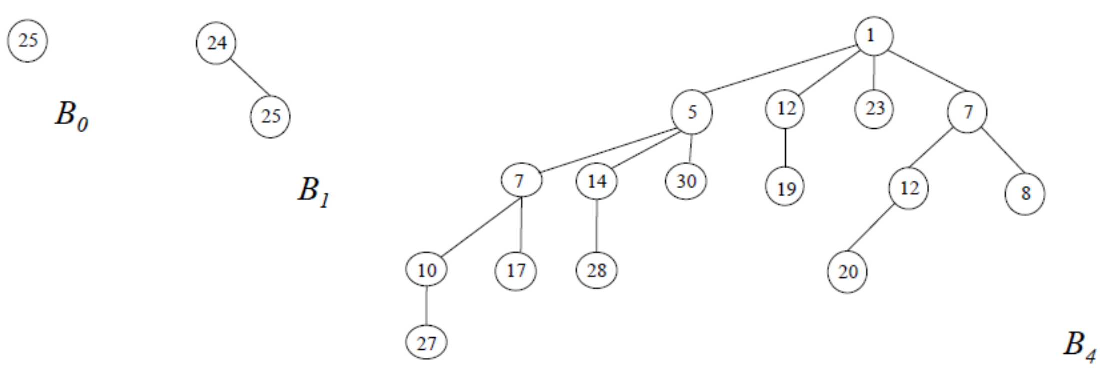
After insert 24:



After insert the first 25:



After insert the second 25:



Problem 9 In a computer game, all the players are divided into a number of groups. Each player can join one group only and is not allowed to join a different group later. Describe an algorithm for checking if two players are in the same group. What is the running time of your algorithm?

Solution: Use the disjoint set union-find data structure with union-by-size and path compression heuristics. The amortized complexity for checking if two players are in the same group is O(log* n).