CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 9
Submit via Gradescope

Name: Xuran Feng
Collaborators:

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

1. ***(25 points)*** *Exercise 17.3-3.(Hint: a reasonable potential function to use is* $\phi(D_i) = kn_i \cdot \ln n_i$ *where* n_i *is the number of elements in the binary heap, and k is a big enough constant. You can use this function and just show the change in potential for each of the two operations.)*

Solution: For the insert operation, the number of elments will increase by 1, so the delta of potential function is $k(n_i + 1) \cdot ln(1 + n_i) - kn_i \cdot lnn_i$, the c_i of insert operation is O(lg n) and the sum of c_i and delta is still O(lg n) because when n is large enough, lnn_i and $ln(n_i + 1)$ are asymptopically same and the delta becomes a constant portion of O(lgn), so the sum is still O(lgn); for the extract-min operation, the number of elments will decrease by 1, so the delta of potential function is $k(n_i - 1) \cdot ln(n_i - 1) - kn_i \cdot lnn_i$, the c_i of extract-min operation is O(lg n) and the sum of c_i and delta becomes O(1) because when k is big enough, the negative constant portion of O(lgn) will cancel off the original O(lgn) of extract-min operation so now it becomes O(1).

2. ***(25 points)*** *Exercise 17.3-6.*

Solution: Use two stacks, where stack1 is only responsible for enqueue and stack2 is only responsible for dequeue. At first, the stack1 and stack2 are both empty, when we begin to enqueue items, stack1 is getting filled and stack2 is still empty. When we dequeue an item from stack2, if the stack2 is empty, we need to move all items in stack1 into stack2 by popping and pushing one by one, then we can call pop() from the stack2; at the same time, if we want to enqueue, we just use stack1 to do the push().

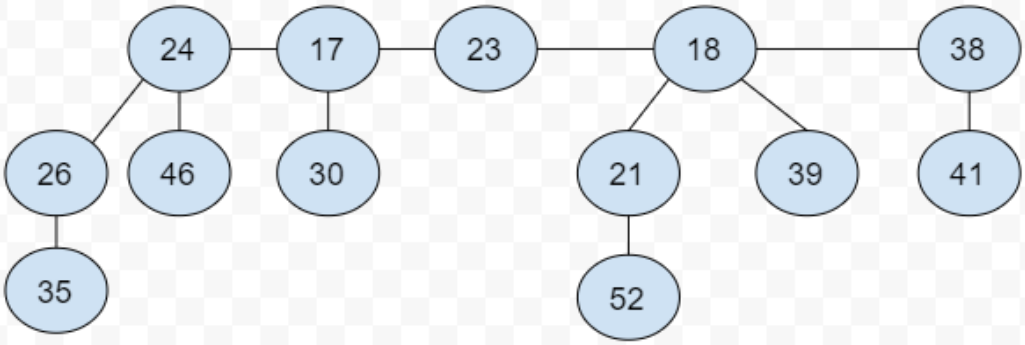
```
let stack1, stack2 be two stacks
ENQUEUE:
    PUSH(stack1,s)
DEQUEUE:
    if(stack2 is empty)
        while(stack1.size>0)
            PUSH(stack2, stack1.POP())
    return POP(stack2)
```

The amortized cost of enqueue and dequeue will be O(1) because we can do a global analysis. Even though the stack1 needs to move all elements into stack2 from time to time when stack2 is empty, each element only needs to get pushed into stack1 one time(ENQUEUE) and popped from stack1 one time, then get pushed into stack2 one time(for move) and popped from stack2 one time(DEQUEUE), so each item only needs to go thourgh constant times of POP() and PUSH(). Since POP() and PUSH() are O(1) amortized cost for stack according to textbook, the DEQUEUE and ENQUEUE are also amortized O(1).

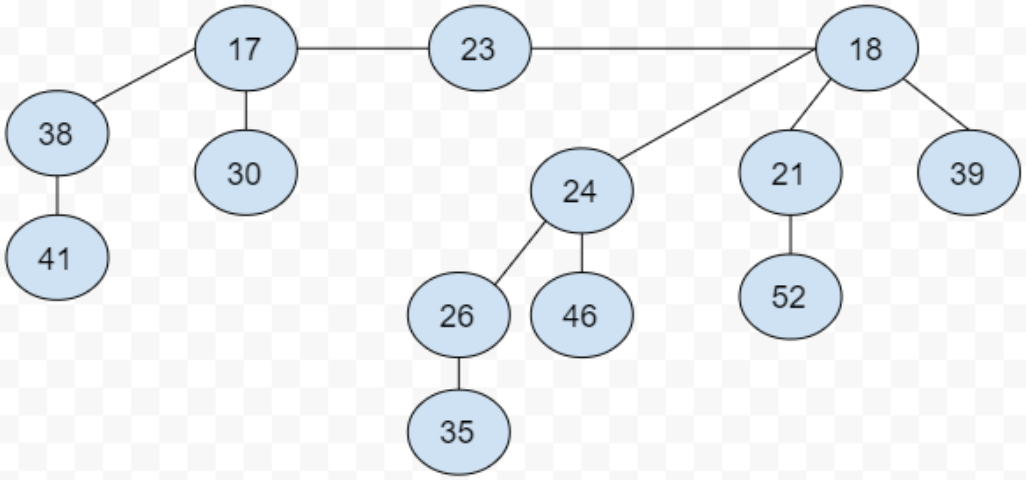
3. ***(25 points)*** *Exercise 19.2-1.*

Solution:

After removing minimum node with key 7



Heaps with same degree will be consolidated.



4. ***(50 points)*** *Implement binomial heaps as described in class and in the book. You should use links (pointers) to implement the structure as shown in the fig .*

```java
import jdk.jshell.spi.SPIResolutionException;

public class BinomialHeap {
    public static void main(String[] args) throws Exception {
        BinomialHeap myHeap=new BinomialHeap();
        BinomialHeap myHeap2=new BinomialHeap();
        int a[] = {12, 7, 25, 15, 28, 33, 41};
        int b[] = {27,11,8,17,14,38,6,29,12,18,1,25,10 };
        //for(Integer i:a) myHeap.insert(i);
        for(Integer i:b) myHeap2.insert(i);
        //myHeap.print();
        //myHeap2.print();
        //myHeap2.decreaseKey(38,5);
        //myHeap2.print();
        System.out.println(myHeap2.getMinimum().key);
        System.out.println(myHeap2.extractMin().key);
        //myHeap2.print();
        System.out.println(myHeap2.extractMin().key);
        System.out.println(myHeap2.extractMin().key);
        //myHeap2.print();
        //myHeap2.delete(18);
        //myHeap.union(myHeap2).print();
    }

    private class heapNode{
        int key;
        int degree;
        heapNode parent,child,sibling;

        public heapNode(int key){
            this.key=key;
            this.degree=0;
            parent=child=sibling=null;
        }
    }

    private heapNode head;
    public BinomialHeap(){
        this.head=null;
    }

    public BinomialHeap(heapNode node){
        this.head=node;
    }

    public BinomialHeap makeHeap(heapNode node){
        return new BinomialHeap(node);
    }

    public heapNode getMinimum() throws Exception {
        int ans=Integer.MAX_VALUE;
        heapNode minimum=null;
        heapNode iter=head;
        while(iter!=null){
            if(ans>iter.key){
                ans=iter.key;
                minimum=iter;
            }
            iter=iter.sibling;
```

```
    }
    if(minimum==null) throw new Exception("the binomial heap is null");
    return minimum;
}

public BinomialHeap union(BinomialHeap heap){
    head=union(head,heap.head);
    return makeHeap(head);
}

private heapNode union(heapNode node1, heapNode node2) {
    heapNode dummy=new heapNode(-1);
    dummy.sibling=mergeTwoLists(node1,node2);
    heapNode iter=dummy.sibling;
    heapNode next=iter.sibling,nnext,prev=dummy;
    while(next!=null){
        nnext=next.sibling;
        if(next.degree<iter.degree){
            prev.sibling=next;
            next.sibling=iter;
            iter.sibling=nnext;
            prev=next;
            next=nnext;
            continue;
        }
        if(next.degree!=iter.degree){
            iter=next;
        }
        else{
            iter.sibling=null;
            next.sibling=null;
            if(iter.key>=next.key) iter=incorporate(iter,next);
            else iter=incorporate(next,iter);
            prev.sibling=iter;
            iter.sibling=nnext;
        }
        next=nnext;
    }
    return dummy.sibling;
}
//merge two heads in ascending degree
private heapNode mergeTwoLists(heapNode node1, heapNode node2){
    if(node1==null) return node2;
    if(node2==null) return node1;
    if(node1.degree<=node2.degree){
        node1.sibling=mergeTwoLists(node1.sibling,node2);
        return node1;
    }
    node2.sibling=mergeTwoLists(node1,node2.sibling);
    return node2;
}
private heapNode incorporate(heapNode node1, heapNode node2){
    node1.sibling=node2.child;
    node1.parent=node2;
    node2.child=node1;
    node2.degree++;
    return node2;
}

public void insert(int key) throws Exception {
```

```
    heapNode newNode=new heapNode(key);
    if(find(key)) throw new Exception("Cannot add same key into the binomial heap!");
    head=union(newNode,head);
}

public heapNode extractMin() throws Exception{
    heapNode minimum=getMinimum();
    heapNode iter,dummy=new heapNode(-1);
    dummy.sibling=head;
    iter=dummy;
    while(iter.sibling!=minimum){
        iter=iter.sibling;
    }
    iter.sibling=minimum.sibling;
    minimum.sibling=null;
    head=union(head,disassemble(minimum.child));
    return minimum;
}

private heapNode disassemble(heapNode minimum){
    if(minimum==null) return null;
    heapNode curr=minimum,next=curr.sibling,nnext;
    curr.sibling=null;
    while(next!=null){
        nnext=next.sibling;
        curr.parent=null;
        next.sibling=curr;
        curr=next;
        next=nnext;
    }
    curr.parent=null;
    return curr;
}

public void delete(int key) throws Exception {
    decreaseKey(key,Integer.MIN_VALUE);
    extractMin();
}

public void decreaseKey(int key, int value) throws Exception {
    if(!find(key)) throw new Exception("no key found!");
    heapNode iter=head;
    while(iter!=null){
        if(iter.key==key){iter.key=value;return;}
        if(iter.key>key){
            iter=iter.sibling;
            continue;
        }
        heapNode node=dfs(iter.child,key),parent;
        if(node!=null) {
            node.key=value;
            parent=node.parent;
            while(parent!=null){
                if(node.key>parent.key) break;
                int tmp=node.key;
                node.key=parent.key;
                parent.key=tmp;
                node=parent;
                parent=parent.parent;
            }
```

```
            break;
        }
        iter=iter.sibling;
    }
}

private boolean find(int key){
    heapNode iter=head;
    while(iter!=null){
        if(iter.key==key) return true;
        if(iter.key>key){
            iter=iter.sibling;
            continue;
        }
        if(dfs(iter.child,key)!=null) return true;
        iter=iter.sibling;
    }
    return false;
}
private heapNode dfs(heapNode node,int key){
    if(node==null) return null;
    if(node.key==key) return node;
    heapNode node1=dfs(node.child,key);
    if(node1==null) node1=dfs(node.sibling,key);
    return node1;
}

public void print() {
    if (head == null)
        return ;

    heapNode p = head;
    System.out.printf("== Binomial Heap( ");
    while (p != null) {
        System.out.printf("B%d ", p.degree);
        p = p.sibling;
    }
    System.out.printf(")details: \n");

    int i=0;
    p = head;
    while (p != null) {
        i++;
        System.out.printf("%d. BinaryHeapB%d: \n", i, p.degree);
        System.out.printf("\t%2d(%d) is root\n", p.key, p.degree);

        print(p.child, p, 1);
        p = p.sibling;
    }
    System.out.printf("\n");
}
private void print(heapNode node, heapNode prev, int direction) {
    while(node != null)
    {
        if(direction==1)
            System.out.printf("\t%2d(%d) is %2d's child\n", node.key, node.degree, prev.key);
        else
            System.out.printf("\t%2d(%d) is %2d's next\n", node.key, node.degree, prev.key);
```

```
            if (node.child != null)
                print(node.child, node, 1);


            prev = node;
            node = node.sibling;
            direction = 2;
        }
    }
}
```

Your implementation should include the operations: Make-heap, Insert, Minimum, Extract-Min, Union, Decrease-Key, Delete

Make sure to preserve the characteristics of binomial heaps at all times:
(1) each component should be a binomial tree with children-keys bigger than the parent-key;

(2) the binomial trees should be in order of size from left to right. Test your code several arrays set of random generated integers (keys).

Solution:

5. ***(Extra Credit)*** *Find a way to nicely draw the binomial heap created from input, like in the figure.*

Solution:

6. ***(Extra Credit)*** *Write code to implement Fibonacci Heaps, with discussed operations:ExtractMin, Union, Consolidate, DecreaseKey, Delete.*

Solution:

7. ***(Extra Credit)*** *Figure out what are the marked nodes on Fibonacci Heaps. In particular explain how the potential function works for FIB-HEAP-EXTRACT-MEAN and FIB-HEAP-DECREASE-KEY operations.*

Solution: