CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 9
Submit via Gradescope

Name: Shri Datta Madhira (NUID: 001557772)
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 $\phi(D_i) = kn_i \ln n_i$ 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:
Given, $\phi(D_i) = kn_i \ln n_i$ where n_i is number of elements in the heap. INSERT and EXTRACT-MIN operations will take $O(\lg n)$ worst-case time.

THE TRICK (as explained by the professor): $\lim_{n\to\infty} n_i \ln(\frac{n_i}{n_i-1}) = 1 or < 2.$

INSERT operation inserts an element into the heap. So, the number of elements after the operation are 1 more than the number of elements before the operation. So, $\phi(D_i) = k(n_i)\ln(n_i)$ and $\phi(D_{i-1}) = k(n_i - 1)\ln(n_i - 1)$. So, the amortized cost of INSERT operation is,

$$\hat{c_i} = c_i + k(n_i)\ln(n_i) - k(n_i - 1)\ln(n_i - 1)$$
$$\hat{c_i} = c_i + k\ln(n_i - 1) + kn_i\ln(n_i) - k(n_i - 1)\ln(n_i - 1)$$
$$\hat{c_i} = k\ln n_i + k\ln(n_i - 1) + kn_i\ln(\frac{n_i}{n_i - 1})$$
$$\hat{c_i} \leq 2k\ln n_i + kn_i\ln(1 + \frac{1}{n_i - 1})$$
$$\hat{c_i} \leq 2k\ln n_i + k\ln(1 + \frac{1}{n_i - 1})^{n_i}$$
$$\hat{c_i} < 2k\ln n_i + 2k < 2k(\ln n_i + 1)$$
$$\boldsymbol{\hat{c_i} = \Theta(\ln n_i)}$$

EXTRACT-MIN operation extracts the minimum element from the heap. So, the number of elements after the operation are 1 less than the number of elements before the operation. So, $\phi(D_i) = k(n_i - 1)\ln(n_i - 1)$ and $\phi(D_{i-1}) = k(n_i)\ln(n_i)$. So, the amortized cost of EXTRACT-MIN operation is,

$$\hat{c_i} = c_i + k(n_i - 1)\ln(n_i - 1) - k(n_i)\ln(n_i)$$
$$\hat{c_i} = k\ln n_i + kn_i\ln(n_i - 1) - kn_i\ln(n_i) - k\ln(n_i - 1)$$
$$\hat{c_i} = k\ln n_i - k\ln(n_i - 1) + kn_i\ln(\frac{n_i - 1}{n_i})$$
$$\hat{c_i} = k\ln(\frac{n_i}{n_i - 1}) + kn_i\ln(\frac{n_i - 1}{n_i})$$
$$\hat{c_i} < 2k + kn_i\ln(\frac{n_i - 1}{n_i}) < 2k - k\ln(\frac{n_i - 1}{n_i})^{-n_i} < 2k - k\ln(1 + \frac{1}{n_i - 1})^{n_i}$$
$$\hat{c_i} < 2k - k < k$$
$$\boldsymbol{\hat{c_i} = \Theta(1)}$$

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

Solution:

Algorithm 3: DEQUEUE

```
Function QUEUE-2-STACK(Q):
    S1[1...n], S2[1...n]
    ENQUEUE(x,S1,S2)
    DEQUEUE(S1,S2)
================================================================
Function QUEUE-2-STACK-ENQUEUE(x,S1,S2):
    S2.top = S2.top + 1
    S2[S2.top] = x
================================================================
Function QUEUE-2-STACK-DeQUEUE(S1,S2):
    If STACK-EMPTY(S1) and STACK-EMPTY(S2) :
        Return "UNDERFLOW"
    If STACK-EMPTY(S1) :
        While !STACK-EMPTY(S2):
            S1.top = S1.top + 1
            S2.top = S2.top - 1
            S1[S1.top] = S2[S2.top+1]
        Return S1[S1.top]
```

As we can see from the algorithm above, we are using two stacks to implement the Queue. For the Enqueue operation, we will push the elements into Stack-2. But as stack follows LIFO, we will use a second stack for the Dequeue operation. For the Dequeue operation, we will empty stack-2, push all the elements into stack-1. This operation reverses the order of the elements, thus giving us the element that is first inserted at the top of stack-1. This makes it easy to pop that element mirroring a queue.

As we can observe, the PUSH and POP operations for the stacks take O(1) time. Because, we are pushing to the top of the stack and we are popping from the top of the stack. So, the actual cost (c_i) of ENQUEUE is 1, and DEQUEUE is 3. The potential function $\phi(D_i) = n_i$ where n_i is number of elements in the stack.

ENQUEUE: This is an insert operation which means the number of elements after the operation are 1 more than the number of elements before the operation. So, the amortized cost of this operation will be,

$$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + n_i - (n_i - 1) = 1 + 1 = 2$$

Therefore, the emortized cost of ENQUEUE is $O(1)$.

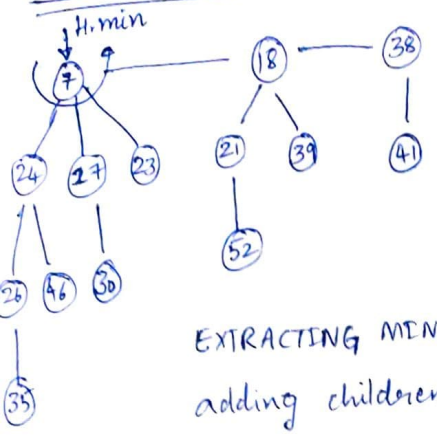
DEQUEUE: This is a get operation which means the number of elements after the operation are 1 less than the number of elements before the operation. So, the amortized cost of this operation will be,

$$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}) = 3 + (n_i - 1) - n_i = 1 - 1 = 0$$

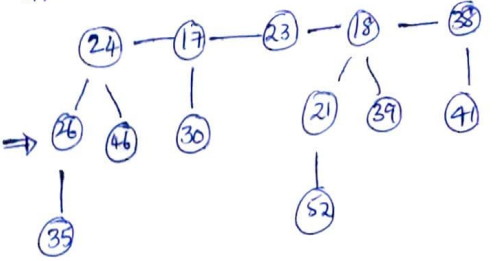
Therefore, the emortized cost of DEQUEUE is $O(1)$.

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

INITIAL TREE :-

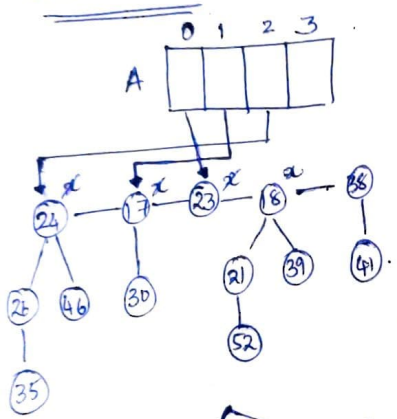


H :-



EXTRACTING MINIMUM;
adding children of min to root list.

CONSOLIDATE (H) :-



1) A[2] ≠ NIL (x)
A[2] = 24

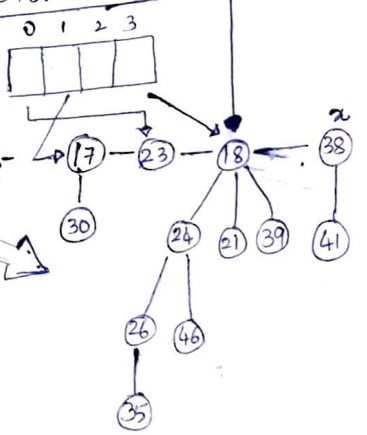
2) A[1] ≠ NIL (x)
A[1] = 17

3) A[0] ≠ NIL (x)
A[0] = 23

4) A[2] ≠ NIL (✓)
y = 24.
18 > 24 ⇒ (x)
Heap-Link (y, x)
24, 18
A[2] = NIL
d = 3

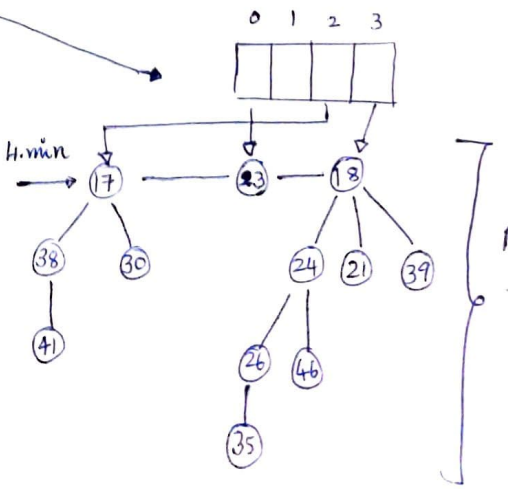
A[3] ≠ NIL (x)
A[3] = 18.

Changes due to Iteration #4.



5) A[1] ≠ NIL (✓)
y = 17
38 > 17 (✓)
Exchange x & y
38 & 17.
Heap-Link (H, 38, 17)
A[1] = NIL
d = 2

A[2] ≠ NIL (x)
A[2] = 17.



FINAL HEAP.

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 figure 1. 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:

```
#include <iostream>
#include <cstdlib>
#include <queue>

using namespace std;

struct node{
    int value;
    int degree;
    node* parent;
    node* child;
    node* sibling;
};

class BinomialHeap{
private:
    node *H;
    node *Hr;
public:
    node* Make_heap();
    node* init_node(int);
    void link(node*, node*);
    node* Union(node*, node*);
    node* Insert(node*, node*);
    node* Merge(node*, node*);
    node* Extract_Min(node*);
    node* Minimum(node*);
    void Reverse_list(node*);
    void print(node*);
    node* Search(node*, int);
    void decrease_key(node*, int, int);
    void Delete(node*, int);

    BinomialHeap(){
        H = Make_heap();
        Hr = Make_heap();
    }
```

```
};

// Initialize Heap
node* BinomialHeap::Make_heap(){
    node* head;
    head = NULL;
    return head;
}

// Linking nodes in Binomial Heap
void BinomialHeap::link(node* n1, node* n2){
    n1->parent = n2;
    n1->sibling = n2->child;
    n2->child = n1;
    n2->degree = n2->degree + 1;
}

// Create Nodes in Binomial Heap
node* BinomialHeap::init_node(int val){
    node* newNode = new node;
    newNode->value = val;
    newNode->parent = NULL;
    newNode->child = NULL;
    newNode->sibling = NULL;
    newNode->degree = 0;
    return newNode;
}

// Insert Nodes in Binomial Heap
node* BinomialHeap::Insert(node* H, node* n){
    node* H1 = Make_heap();
    H1 = n;
    H = Union(H, H1);
    return H;
}

// Union Nodes in Binomial Heap
node* BinomialHeap::Union(node* H1, node* H2){
    node *H = Make_heap();

    H = Merge(H1, H2);
    if (H == NULL){
        return H;
    }
```

```
    node* prev;
    node* next;
    node* curr;

    prev = NULL;
    curr = H;
    next = curr->sibling;
    while (next != NULL){
        if ((curr->degree != next->degree) || ((next->sibling != NULL)
            && (next->sibling)->degree == curr->degree)){
            prev = curr;
            curr = next;
        }
        else{
            if (curr->value <= next->value){
                curr->sibling = next->sibling;
                link(next, curr);
            }
            else{
                if (prev == NULL){
                    H = next;
                }
                else{
                    prev->sibling = next;
                }
                link(curr, next);
                curr = next;
            }
        }
        next = curr->sibling;
    }
    return H;
}

// Merge Nodes in Binomial Heap
node* BinomialHeap::Merge(node* H1, node* H2){
    node* H = Make_heap();
    node* y;
    node* z;
    node* a;
    node* b;

    y = H1;
    z = H2;
```

```
    if (y != NULL){
        if (z != NULL){
            if (y->degree <= z->degree){
                H = y;
            }
            else if (y->degree > z->degree){
                H = z;
            }
        }
        else{
            H = y;
        }
    }
    else{
        H = z;
    }

    while (y != NULL && z != NULL){
        if (y->degree < z->degree){
            y = y->sibling;
        }
        else if (y->degree == z->degree){
            a = y->sibling;
            y->sibling = z;
            y = a;
        }
        else{
            b = z->sibling;
            z->sibling = y;
            z = b;
        }
    }
    return H;
}

// Display Binomial Heap
//void BinomialHeap::print(node* H){
//    if (H == NULL){
//        cout << "The Heap is empty" << endl;
//    }
//    else {
//        cout << "The root nodes are:" << endl;
//        node* p;
//        p = H;
//
```

```
//         cout << H->value;
//
//         while(p != NULL){
//             cout << p->value;
//             if(p->child != NULL){
//                 cout << "-->";
//             }
//             p = p -> child;
//         }
//
//         while (p != NULL){
//             cout << p->value;
//             if (p->sibling != NULL){
//                 cout<<"-->";
//             }
//             p = p->sibling;
//         }
//         cout << endl;
//     }
//}
void BinomialHeap::print(node* H){
    if (H == NULL){
        cout << "The Heap is empty" << endl;
    }
    node* currPtr = H;
    while (currPtr != nullptr) {
        cout<<"B"<<currPtr->degree<<endl;
        cout<<"There are "<<pow(2, currPtr->degree)<<" nodes in this
           tree"<<endl;
        cout<<"The level order traversal is"<<endl;
        queue<node*> q;
        q.push(currPtr);
        while (!q.empty()) {
            node* p = q.front();
            q.pop();
            cout<<p->value<<" ";

            if (p->child != nullptr) {
                node* tempPtr = p->child;
                while (tempPtr != nullptr) {
                    q.push(tempPtr);
                    tempPtr = tempPtr->sibling;
                }
            }
        }
```

```
        currPtr = currPtr->sibling;
        cout<<endl<<endl;
    }
}

// Extract Minimum
node* BinomialHeap::Extract_Min(node* H1)
{
    Hr = NULL;
    node* t = NULL;
    node* x = H1;

    if (x == NULL){
        cout << "Nothing to Extract" << endl;
        return x;
    }

    int min = x->value;
    node* p = x;
    while (p->sibling != NULL){
        if ((p->sibling)->value < min){
            min = (p->sibling)->value;
            t = p;
            x = p->sibling;
        }
        p = p->sibling;
    }

    if (t == NULL && x->sibling == NULL){
        H1 = NULL;
    }
    else if (t == NULL){
        H1 = x->sibling;
    }
    else if (t->sibling == NULL){
        t = NULL;
    }
    else{
        t->sibling = x->sibling;
    }

    if (x->child != NULL){
        Reverse_list(x->child);
        (x->child)->sibling = NULL;
    }
```

```
    H = Union(H1, Hr);

    return x;
}

// Reverse List
void BinomialHeap::Reverse_list(node* y){
    if (y->sibling != NULL){
        Reverse_list(y->sibling);
        (y->sibling)->sibling = y;
    }
    else{
        Hr = y;
    }
}

// Search Nodes in Binomial Heap
node* BinomialHeap::Search(node* H, int k)
{
    node* x = H;
    node* p = NULL;

    if (x->value == k){
        p = x;
        return p;
    }

    if (x->child != NULL && p == NULL){
        p = Search(x->child, k);
    }
    if (x->sibling != NULL && p == NULL){
        p = Search(x->sibling, k);
    }

    return p;
}

// Decrease key of a node
void BinomialHeap::decrease_key(node* H, int i, int k)
{
    int temp;
    node* p;
    node* y;
    node* z;
    p = Search(H, i);
```

```
    if (p == NULL)

    {
        cout<<"Invalid choice of key"<<endl;
    }
    else{
        if (k > p->value)
        {
            cout<<"Error!! New key is greater than current key"<<endl;
        }
        else{
            p->value = k;
            y = p;
            z = p->parent;

            while (z != NULL && y->value < z->value)
            {
                temp = y->value;
                y->value = z->value;
                z->value = temp;
                y = z;
                z = z->parent;
            }
            cout<<"Key reduced successfully"<<endl;
        }
    }
}

// Delete Nodes in Binomial Heap
void BinomialHeap::Delete(node* H, int k)
{
    node* np;

    if (H == NULL)
    {
        cout<<"\nHEAP EMPTY!!!!!";
    }
    else{
        decrease_key(H, k, -1000);
        np = Extract_Min(H);
        if (np != NULL){
            cout<<"Node Deleted Successfully"<<endl;
        }
        else{
```

```
            cout << "Some error occured in Delete." << endl;
        }
    }
}

node* BinomialHeap::Minimum(node* h){
    int min_val = 999999;
    node* curr = h;
    node* min = nullptr;

    while (curr != nullptr) {
        if (curr->value < min_val) {
            min_val = curr->value;
            min = curr;
        }
        curr = curr->sibling;
    }
    return min;
}

int main()
{
    int choice;
    BinomialHeap bh;
    node* n;
    node *H;

    H = bh.Make_heap();

    while(1){
        cout << "1.INsert || 2.Extract_Min || 3.decrease key || 4.
           delete || 5. Minimum || 6.print || 0.exit" << endl;
        cout << "Enter your choice:";
        cin >> choice;
        switch(choice){
            case 1:
            {
                int val;
                cout << "Enter the value of the element you want to
                   insert:";
                cin >> val;
                n = bh.init_node(val);
                H = bh.Insert(H, n);
                break;
            }
```

```
            case 2:
                n = bh.Extract_Min(H);
                if (n != nullptr) {
                    cout << "Result of Extract Min: " << n -> value <<
                       endl;
                } else {
                    cout << "Heap is empty." << endl;
                }
                break;
            case 3:
            {
                int val, new_val;
                cout << "Enter the key to be decreased:";
                cin >> val;
                cout << "Enter new value:";
                cin >> new_val;
                bh.decrease_key(H, val, new_val);
                break;
            }
            case 4:
            {
                int val;
                cout << "Enter key to be deleted:";
                cin >> val;
                bh.Delete(H, val);
                break;
            }
            case 5:
            {
                node* v = bh.Minimum(H);
                cout << "The minimum of the tree is:" << v->value <<
                   endl;
                break;
            }
            case 6:
            {
                bh.print(H);
                break;
            }
            case 0:
                exit(0);
        }
    }
    return 0;
}
```

EXPECTED OUTPUT:

```
1.INsert || 2.Extract_Min || 3.decrease key || 4.delete || 5. Minimum
   || 6.print || 0.exit
Enter your choice:1
Enter the value of the element you want to insert:3
1.INsert || 2.Extract_Min || 3.decrease key || 4.delete || 5. Minimum
   || 6.print || 0.exit
Enter your choice:1
Enter the value of the element you want to insert:1
1.INsert || 2.Extract_Min || 3.decrease key || 4.delete || 5. Minimum
   || 6.print || 0.exit
Enter your choice:1
Enter the value of the element you want to insert:5
1.INsert || 2.Extract_Min || 3.decrease key || 4.delete || 5. Minimum
   || 6.print || 0.exit
Enter your choice:1
Enter the value of the element you want to insert:4
1.INsert || 2.Extract_Min || 3.decrease key || 4.delete || 5. Minimum
   || 6.print || 0.exit
Enter your choice:1
Enter the value of the element you want to insert:10
1.INsert || 2.Extract_Min || 3.decrease key || 4.delete || 5. Minimum
   || 6.print || 0.exit
Enter your choice:1
Enter the value of the element you want to insert:15
1.INsert || 2.Extract_Min || 3.decrease key || 4.delete || 5. Minimum
   || 6.print || 0.exit
Enter your choice:6
B1
There are 2 nodes in this tree
The level order traversal is
10 15
B2
There are 4 nodes in this tree
The level order traversal is
1 4 3 5
1.INsert || 2.Extract_Min || 3.decrease key || 4.delete || 5. Minimum
   || 6.print || 0.exit
Enter your choice:5
The minimum of the tree is:1
1.INsert || 2.Extract_Min || 3.decrease key || 4.delete || 5. Minimum
   || 6.print || 0.exit
Enter your choice:2
Result of Extract Min: 1
```

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:Extract Min, Union, Consolidate, Decrease Key, 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: