CS225 Assignment3 | Group 10

--***Exercise 1***--

(i)

Fake code of $siftUp(n)$:

$siftUp(n)$:
$parent = \lfloor \frac{n}{2} \rfloor$
$while\ \lfloor parent/2 \rfloor > 1$:
$If\ parent \neq Max[parent, leftChild, rightChild]$:
$Switch\ parent\ with\ Max[parent, leftChild, rightChild]$
$siftUp(parent)$

For the worst case, n will go to 1, so the time complexity is $O(log\ n)$ since the heap degree is $log\ n$.

So the total complexity of insert is $O(1 + log\ n) = O(log\ n)$.

(ii)

The $siftDown$ is now changed into two parts:

1.First change the value of the $siftDown$ node to be $-\infty$, and by using the normal $siftdown$ method, this node will be moved to bottom level of the heap (a leaf node, but may not the last leaf node). In this process by changing the $siftDown$ algorithm, the comparisons can be $logn$ (EX. add a if condition by check whether the compared value is $-\infty$, if it is , exchange the biggest child with this parent node of value $-\infty$. Thus just one time comparison is exacuted.)

2. Now the $siftDown$ path is made, the second thing is to find where this node should be placed along the $siftDown$ path, in other words, we need to determine which heap level the node should be placed. By using the binary search, first check the middle level of the heap. Then check iteratively the middle level of the previous level. We can find the target level with $O(lglgn)$, ($T(m) = T(\frac{m}{2}) + \theta(1), m = lgn$).

So the total complexity is $lgn + O(lglgn)$

--***Exercise 2***--

Please refer to the program and pdf files in "Q2" to recover our choice of the associative array and the implementation.

--***Exercise 3***--

3.1

Note "prev" here is a pointer to next older sibling; "next" here is a pointer to next younger

$Find\_Min$

> This operation returns the minimum node of the heap structure.

```
int PairHeap::find_min(){
    return min;
}
```

$Merge$

> This operation "combines" two heap structures.
> Specifically, it does the following:
>
> 1. Add the tree with a larger root value as the last child of another tree.
> 2. Delete the tree with a larger root from the root list.

```
Pairheap* PairHeap::Merge(Pairheap* heap1,Pairheap* heap2){
    // if one of the heaps are empty, return the other heap.
    if (heap1 == NULL){
        return heap2;
    }
    else if (heap2 == NULL){
        return heap1;
    }
    // put the heap with a larger root value as the last child of the heap w
    Pairheap * temp_ptr;
    if (heap1->key > heap2->key){
        temp_ptr = heap1->child;

        // delete heap2 from the root list
        if(heap2->prev){
            heap2->prev->next = heap2->next;
            heap2->next->prev = heap2->prev;
        }else{
            heap2->next->prev = NULL;
        }

        if(temp_ptr){
            // traverse through the child list to find the last child
            while (temp_ptr->next != NULL) temp_ptr = temp_ptr->next;
            temp_ptr->next = heap2;
            // maintain the definition of the heap nodes
            heap2->next = heap1;
            heap2->prev = temp_ptr;
        }else{
            heap1->child = heap2;
        }
        // delete heap2 from the root list
        if(heap2->prev){
            heap2->prev->next = heap2->next;
            heap2->next->prev = heap2->prev;
        }
        return heap1;
    }else{
        temp_ptr = heap2->child;

        // delete heap1 from the root list
```

```
        // delete heap1 from the root list
        if(heap1->prev){
            heap1->prev->next = heap1->next;
            heap1->next->prev = heap1->prev;
        }else{
            heap1->next->prev = NULL;
        }

        if(temp_ptr){
            // traverse through the child list to find the last child
            while (temp_ptr->next != NULL) temp_ptr = temp_ptr->next;
            temp_ptr->next = heap1;
            // maintain the definition of the heap nodes
            heap1->next = NULL;
            heap1->prev = temp_ptr;
        }else{
            heap2->child = heap1;
            heap1->prev = heap2;
            heap1->next = NULL;
        }
        return heap2;
    }
}
```

Insert

> 1. add the node to the root list
> 2. update h.min

```
void PairHeap::Insert(Pairheap* node){

    // Add the node to the beginning of the root list.
    node->prev = NULL;
    node->next = roots;
    roots->prev = node;
    roots = node
    // update minimum
    min = min.key > node.key ? node : min;
}
```

DecreaseKey

> 1. decrease value of the given node (indicated by the handle h) to k
> 2. move the corresponding subtree to the root list, if necessary

```
void Pairheap::DecreaseKey(Pairheap* h, int k){
    // decrease the value
    h->value = k;

    if (h in roots){
        // if the node indicated by the given handle is a root, the steps hc
        return;
    }else{
        //find the parent of this node
```

```
            Pairheap* parent = h;
            while(parent->prev->child != parent)
            parent = parent->prev;

            parent = parent->prev;
            if (parent->key < h->key){
                // steps are complete now, return.
                return;
            }else{
            // move the subtree to the root list
                if(parent->child = h){
                    parent->child = h->next;
                    h->next->prev = parent;
                }else{
                    h->prev->next = h->next;
                    if(h->next != NULL)
                        h->next->prev = h->prev;
                }
            Insert(h, roots);
            }
        }
    }
```

$ExtractMin$

> 1. Delete the min node from the root list
> 2. Add all children of the deleted node to the root list
> 3. Find the new minimum
> 4. Combine
> 5. Return the deleted min node

```
Pairheap* Pairheap::ExtractMin(){
    Pairheap* minnode = min;

    Pairheap* child_ptr = min->child;
    // add all children to the root list
    while(child_ptr != NULL){
        child_ptr = child_ptr->next;
        Insert(child_ptr->prev, roots);
    }
    min->child = NULL;
    if(min == roots){
        min->next->prev = NULL;
        roots = min->next;
    }else{
        min->prev->next = min->next;
        min->next->prev = min->prev;
    }
    min->next = NULL;
    min->prev = NULL;

    // find the new minimum
    min = roots;
    Pairheap* iterator = roots;
```

```
        // traverse through the root list to find the new minimum
        while(iterator != NULL){
            min = min.key > iterator.key ? iterator : min;
            iterator = iterator->next;
        }

        // combine (one pass combination)
        iterator = roots;
        while(iterator != NULL){
            if(iterator->next != NULL){
                // this following step is executed because the next and prev inf
                iterator = iterator->next->next;
                Merge(iterator->prev->prev, iterator-prev);
            }
            iterator = iterator->next;
        }
        return minnode;
    }
```

3.2

1.$FindMin$:

$return\ heap.root$

2.$Merge(heap1.root, heap2.root)$:

If $heap1$ is empty:

return $heap2.root$

If $heap2$ is empty:

return $heap1.root$

If $heap2.root >= heap1.root$:

$heap1.root -> child -> youngestSibling$: $heap2.root$ //point to the heap1.root before

$heap2.root -> youngerSibling$: $heap1.root$

return $heap1.root$

//don't change heap1.root->youngerSibling since it points to null

Else:

$heap2.root -> child -> youngestSibling$: $heap1.root$

//find the youngest sibling, which point to the heap2.root before

$heap1.root -> youngerSibling$: $heap2.root$

return $heap2.root$

//don't change heap2.root->youngerSibling since it points to null

3. $Insert(heap, node)$:

$heap.root -> youngerSibling$: $node$

$node -> youngerSibling$: $null$

4.$DecreaseKey(position, key)$:

If $position = heap.root$:

$position.value = key$

Else:

For $node$ in $heap$:

If $node->youngerSibling = position \backslash or\ node->child = position$

$node->youngerSibling : position->youngerSibling \backslash or$
$node->child : position->youngerSibling$

//remove from the tree

For $root$ in $heap.rootlist$:

If $root->youngerSibling = null$:

$root->youngerSibling : position$

$position->youngerSibling : null$

$position.value = key$

5.$DeleteMin()$:

For $root$ in $heap.rootlist$:

$Minroot = root\ with\ the\ smallest\ root.value$

$DecreaseKey(Minroot->child, 0)$ //move the child to the rootlist

For $node$ in $heap.rootlist$

If $node->youngerSibling = Minroot$:

$node->youngerSibling = Minroot->youngerSibling$ //remove from the list

For $Node\ 1, Node\ 2$ in $heap.rootlist$:

$Merge(Node\ 1, Node\ 2)$ //Combine all nodes in rootlist

--***Exercise 4***--

Preparations for the exercise:

> INSERT and UNION will have changed run time due to $\mathrm{consolidate}$.
>
> For INSERT, the complexity is bounded by $O(\log(n))$
>
> For UNION, the complexity is bounded by $O(Max\{\log n, \log m\})$ if we denote the sizes of the two heaps by n and m.
>
> For MIN, the complexity is O(1)
>
> For $\mathrm{ExtractMIN}$, the complexity is given by $O(\log(n))$ according to our textbook.

Amortized Analysis:

Note:

- In our consideration, the operation of UNION is not counted.
- R represents the size of the root list, n represents the size of the heap structure.

$Insert$ (just forget about the following content if incorrect)

The worst case happens when a node is inserted into the heaps $root\_i.degree = i$ for all roots in the heap structure, which leads to a "collapse" of the heaps (all heaps are consolidated into one giant heap). The complexity in this case is obviously $O(R) = O(\log n)$.

If we only consider insertions, then the total complexity of insertions for a McGee heap is upper bounded by

$$\sum_{i=1}^{R} \sum_{j=1}^{i} (\log j + 1)$$

For amortized complexity,

$$\begin{aligned} Insert(node) &= O(\frac{\sum_{i=1}^{\log n} \sum_{j=1}^{i} (\log j + 1)}{n}) \\ &= O(\frac{\log \prod_{i=1}^{\log n} i!}{n} + \frac{\log n(\log n + 1)}{n}) \end{aligned}$$