COMP305 Problem Set 3

Exercise 3-2

Modify the $vEB\text{–}tree$ so that each base $u = 2$ structure stores how many times a given key was inserted. So if $V.min \neq NIL$, $V.min\text{–}count$ is how many instances of a given key is in store. Similarly for $V.max$ and for $V.max\text{–}count$. When inserting, update the appropriate counter. When deleting, decrease the counter, unless $count = 0$, in which case return immediately, so as not to attempt to delete a key that is not there.

```python
class VEB:
    u: int
    min: Optional[int]
    max: Optional[int]
    clusters: Optional['List[VEB]']
    summary: Optional['VEB']

    # new attributes
    min_count: int
    max_count: int

    # ... other code

    def insert(self: 'VEB', x: int, count=1):
        # if tree is empty, min = max = x
        if self.min is None:
            self.min = self.max = x
            self.min_count = self.max_count = count
            return

        # increment if x is there
        if x == self.min:
            self.min_count += count
        if x == self.max:
            self.max_count += count

        # swap x and min
        if x < self.min:
            self.min, x = x, self.min
            self.min_count, count = count, self.min_count

        if self.u > 2:
            # if cluster is empty, insert into summary
            if self.cluster_of(x).min is None:
                self.summary.insert(self.high(x))
            self.cluster_of(x).insert(self.low(x), count)

        # update max
        if x > self.max:
            self.max = x
            self.max_count = count

    def delete(self: 'VEB', x: int):
        # similar modifications for delete
        pass
```

Exercise 3-3

Modify the $vEB\text{–}tree$ so that $Insert(V, x)$ and other operations now support records or objects x. If the key is a pointer to the object x, choose $u = 2^{sizeof(pointer)}$ and store keys only. No further change is necessary.

If x is a record, consisting of a key and its associated satelite data as a pointer, then for each inserted key, also keep track of its associated pointer, $V.min\text{–}pointer$ and $V.max\text{–}pointer$. Now, we can define methods to access the satellite data using the key.

```
Key = int
Pointer = object

class Record:
    key: Key
    value: Pointer

class VEB:
    u: int
    clusters: Optional['list[VEB]']
    summary: Optional['VEB']

    # new and modified attributes
    min: Optional[Key]
    max: Optional[Key]
    min_value: Optional[Pointer]
    max_value: Optional[Pointer]

    # ... other code

    def insert(self: 'VEB', x: Record):
        key = x.key
        value = x.value
        # ...

    def delete(self: 'VEB', key: Key):
        # ...
        pass

    def get(self: 'VEB', x: Key):
        if x == self.min:
            return self.min_value
        elif x == self.max:
            return self.max_value
        elif self.u == 2:
            raise KeyError
        return self.cluster_of(x).get(self.low(x))

    # ...
```

Problem 3-1

Part (a)

For simplicity, assume $u = 2^{3^k} = 2^m$. For a $vEB\text{–}tree$ structure with $u^{1/3}$ clusters, the high and low bits of x become $high(x) = \lfloor x/u^{2/3} \rfloor$, $low(x) = x \mathbin{\%} u^{2/3}$, respectively. And we can also index into the structure using $index(x, y) = x \cdot u^{2/3} + y$.

The basic recurrence relation for $vEB\text{–}tree$ operations becomes $T(u) = T(u^{2/3}) + \Theta(1)$. Using change of variable, we can simplify the relation to $S(m) = S(2m/3) + \Theta(1)$. By master theorem, $S(m) \in \Theta(\log m)$ and $T(n) \in \Theta(\log \log u)$, which is on the same asymptotic order of growth. Recursive relations are respected when $u = u^{1/3} \cdot u^{2/3}$, and the code need not change.

1. Operations $Minimum(V, x)$ and $Maximum(V, x)$ are still $\Theta(1)$.
2. $Member(V, x)$ recurses over smaller clusters of $vEB\text{–}tree(u^{2/3})$ structures, described by the above recursion. Thus, $Member(V, x) \in \Theta(\log \log u)$.
3. $Predecessor(V, x)$ and $Successor(V, x)$ recurse either on $V.summary$, which is a $vEB\text{–}tree(u^{1/3})$ structure, or clusters of size $u^{2/3}$, but never both, since calls to $Minimum$ and $Maximum$ are constant-time. Thus, their running time is described by $T(u) = max(T(u^{1/3}), T(u^{2/3})) + \Theta(1)$, which is $O(\log \log u)$.
4. In the same way, calls to $Insert(V, x)$ also recurse over either a structure of size $u^{1/3}$ or $u^{2/3}$, but not both, yielding $O(\log \log u)$.
5. $Delete(V, x)$, as in the original structure, performs one recursive call and one auxiliary call, remaining $O(\log \log u)$.

Part (b)

We will apply lazy propagation to $V.max$ too. Assuming no duplicate values, we can modify the code so that $V.min$ will be symmetric to $V.max$. Complexities remain the same, since the previous recursive relations are not violated. But the code is now much prettier. Assume the following structure for the vEB structure:

```
class VEB:
    u: int
    min: Optional[int]
    max: Optional[int]
    clusters: Optional['List[VEB]']
    summary: Optional['VEB']

    def cluster_of(self, x: int) -> 'VEB':
        return self.clusters[self.high(x)]

    def cluster(self, i: int) -> 'VEB':
        return self.clusters[i]

    def is_empty(self: 'VEB') -> bool:
        return self.min is None and self.max is None

    def is_singleton(self: 'VEB') -> bool:
        return self.min == self.max
```

$Member(V, x)$

This does not change.

```
def contains(self: 'VEB', x: int) -> bool:
    if x in [self.min, self.max]:
        return True
    elif self.u == 2:
        return False
    else:
        return self.cluster_of(x).contains(self.low(x))
```

$Insert(V, x)$

Insert now has to swap both $V.min$ and $V.max$ with the inserted element.

```
def insert(self: 'VEB', x: int):
    # if tree is empty, insertion is trivial
    if self.is_empty():
        self.min = self.max = x
        return
```

```python
    # if tree is a singleton, insert x and update min/max
    if self.is_singleton():
        if x > self.min:
            self.max = x
        else:
            self.min = x
        return

    # if x is less than min, swap x and min
    if x < self.min:
        self.min, x = x, self.min

    # if x is greater than max, swap x and max
    # symmetric to self.min
    if x > self.max:
        self.max, x = x, self.max

    # if cluster is empty, insert it into summary
    if self.cluster_of(x).is_empty():
        self.summary.insert(self.high(x)) # this is still O(1)

    # insert x into appropriate cluster
    self.cluster_of(x).insert(self.low(x))
```

$Delete(V, x)$

The code for delete is now greatly simplified. In addition to $V.min$, $Delete(V, x)$ has to handle $V.max$ as a special case too.

```python
def delete(self: 'VEB', x: int):
    # if tree has one element, empty the tree
    if self.is_singleton():
        self.min = self.max = None
        return

    # if tree has two elements, delete x
    # if x was min, set min to max, else set max to min
    if self.summary.is_empty():
        if x == self.min:
            self.min = self.max
        else:
            self.max = self.min
        return

    # if x is min, it does not appear in any cluster
    # find minimum that DOES appear and replace x with that
    # then mark x so that it is deleted
    if x == self.min:
        first_cluster = self.summary.min
        x = self.index(first_cluster, self.cluster(first_cluster).min)
        self.min = x

    # symmetric to self.min
    if x == self.max:
        last_cluster = self.summary.max
        x = self.index(last_cluster, self.cluster(last_cluster).max)
        self.max = x

    # delete x from appropriate cluster
    # if cluster is empty, delete it from summary
```

```python
    cluster = self.cluster_of(x)
    cluster.delete(self.low(x))
    if cluster.is_empty():
        self.summary.delete(self.high(x))
```

$Successor(V, x)$

Since $V.max$ is now a special attribute, the code for $Successor(V, x)$ has to explicitly check for it just like in $Predecessor(V, x)$.

```python
def successor(self, x: int) -> Optional[int]:
    # if it's the base case, return either self.max or None
    if self.u == 2:
        if x == 0 and self.max == 1:
            return 1
        return None

    # if x is less than min, return min
    # min is a special case because it's not in the tree
    if self.min is not None and x < self.min:
        return self.min

    # if x is less than max of its cluster, find successor in cluster
    max_low = self.cluster_of(x).max
    if max_low is not None and self.low(x) < max_low:
        offset = self.cluster_of(x).successor(self.low(x))
        return self.index(self.high(x), offset)

    # otherwise look for a non-empty cluster in the summary
    succ_cluster = self.summary.successor(self.high(x))
    if succ_cluster is None:
        # symmetric to self.min
        if self.max is not None and x < self.max:
            return self.max
        return None

    # if there is one, find the min of that cluster and return it
    offset = self.cluster(succ_cluster).min
    return self.index(succ_cluster, offset)
```

$Predecessor(V, x)$

Predecessor does not change, as it was already handling the special attribute $V.min$.

```python
def predecessor(self, x: int) -> Optional[int]:
    # if it's the base case, return either self.min or None
    if self.u == 2:
        if x == 1 and self.min == 0:
            return 0
        return None

    # if x is greater than max, return max
    if self.max is not None and x > self.max:
        return self.max

    # if x is greater than min of its cluster,
    # find predecessor in cluster
    min_low = self.cluster_of(x).min
    if min_low is not None and self.low(x) > min_low:
        offset = self.cluster_of(x).predecessor(self.low(x))
```

```
        return self.index(self.high(x), offset)

    # otherwise look for a non-empty cluster in the summary
    pred_cluster = self.summary.predecessor(self.high(x))
    if pred_cluster is None:
        # min is a special case because it's not in the tree
        if self.min is not None and x > self.min:
            return self.min
        return None

    # if there is one, find the max of that cluster and return it
    offset = self.cluster(pred_cluster).max
    return self.index(pred_cluster, offset)
```

Resources

No resources beyond CLRS Chapter 20.3 were used.