AADS - Assignment 5

Aditya Fadhillah (hjg708), Mads Nørregaard (gnz359), Nikolaj Schaltz (bxz911)

January 8, 2025

Contents

1 Exercises from CLRS Chapter 20: Van Emde Boas Trees

1.1 Exercise 20.3-1

Modify vEB trees to support duplicate keys.

Answer:

A standard *van Emde Boas* tree (vEB-tree) maintains a dynamic set of distinct integers from a universe of size U and supports the operations MEMBER, SUCCESSOR, PREDECESSOR, INSERT, and DELETE in $O(\lg \lg U)$ time. To store *duplicates*, we can modify each vEB-tree node to keep a `count` of how many times each key appears, rather than just a simple membership marker.

Insertion (vEB-Insert):

- If a key x is already in the vEB-tree (i.e., $\texttt{count}[x] > 0$), simply increment $\texttt{count}[x]$.
- Otherwise, perform the usual vEB INSERT procedure and set $\texttt{count}[x] = 1$.

```
-------------------------------------------------------------------------------------
vEB-Tree-Insert(V, x)
1   if V.count[x] > 0                    // Key already exists
2       V.count[x] = V.count[x] + 1      // Increment the count
3       return
4   if V.min == NIL
5       vEB-Empty-Tree-Insert(V, x)
6       V.count[x] = 1
7   else if x < V.min
8       exchange x with V.min
9   if V.u > 2
10      if vEB-Tree-Minimum(V.cluster[high(x)]) == NIL
11          vEB-Tree-Insert(V.summary, high(x))
12          vEB-Empty-Tree-Insert(V.cluster[high(x)], low(x))
13          V.cluster[high(x)].count[low(x)] = 1
14      else vEB-Tree-Insert(V.cluster[high(x)], low(x))
15   if x > V.max
16      V.max = x
17   V.count[x] = 1
-------------------------------------------------------------------------------------
```

Deletion (vEB-Delete):

- If $\texttt{count}[x] > 1$, decrement $\texttt{count}[x]$ and stop.
- Otherwise, proceed with the normal vEB DELETE, removing the key entirely (i.e., set $\texttt{count}[x] = 0$).

```
-------------------------------------------------------------------------------------
vEB-Tree-Delete(V, x)
1   // If the key dont exist, do nothing
2   if V.count[x] == 0
3       return

4   // If multiple occurrences exist, just decrement the count
5   if V.count[x] > 1
6       V.count[x] = V.count[x] - 1
7       return

8   // Now handle the single-occurrence case (standard vEB delete)
```

```
9   if V.min == V.max
10      V.min = NIL
11      V.max = NIL
12      V.count[x] = 0
13      return

14  else if V.u == 2
15      if x == 0
16         V.min = 1
17         V.count[0] = 0
18      else
19         V.min = 0
20         V.count[1] = 0
21      V.max = V.min
22      return

23  else if x == V.min
24      first-cluster = vEB-Tree-Minimum(V.summary)
25      x = index(first-cluster, vEB-Tree-Minimum(V.cluster[first-cluster]))
26      V.min = x
27      // Transfer the old count to the new min
28      V.count[x] = V.count[x] + 1
29      // Remove the old x completely
30      V.count[index(first-cluster,
                      vEB-Tree-Minimum(V.cluster[first-cluster]))] = 0

31      vEB-Tree-Delete(V.cluster[high(x)], low(x))
32      if vEB-Tree-Minimum(V.cluster[high(x)]) == NIL
33          vEB-Tree-Delete(V.summary, high(x))
34      if x == V.max
35          summary-max = vEB-Tree-Maximum(V.summary)
36          if summary-max == NIL
37              V.max = V.min
38          else
39              V.max = index(summary-max,
                              vEB-Tree-Maximum(V.cluster[summary-max]))

40  else if x == V.max
41      V.max = index(high(x), vEB-Tree-Maximum(V.cluster[high(x)]))
42      vEB-Tree-Delete(V.cluster[high(x)], low(x))
43      if vEB-Tree-Minimum(V.cluster[high(x)]) == NIL
44          vEB-Tree-Delete(V.summary, high(x))

45  else
46      vEB-Tree-Delete(V.cluster[high(x)], low(x))
47      if vEB-Tree-Minimum(V.cluster[high(x)]) == NIL
48          vEB-Tree-Delete(V.summary, high(x))
```

Membership, Successor, Predecessor:

- MEMBER now checks if $\texttt{count}[x] > 0$.
- SUCCESSOR and PREDECESSOR do not change, because they operate on distinct keys. The multiplicity of a key does not affect its successor or predecessor among distinct values.

```
-------------------------------------------------------------------------------------
vEB-Tree-Member(V, x)
1  // If x has a positive count, it is in the tree.
2  if V.count[x] > 0
3      return TRUE
4  if x == V.min or x == V.max
5      return TRUE
6  else if V.u == 2
7      return FALSE
8  else
9      return vEB-Tree-Member(V.cluster[ high(x) ], low(x))
-------------------------------------------------------------------------------------
```

1.2 Exercise 20.3-2

Modify vEB trees to support keys that have associated satellite data.

Answer:

We can extend a vEB tree so that each key x also has associated *satellite data* (e.g., a record or an object). The main idea is to store a pointer or reference to the satellite data whenever we insert a new key, and to retrieve or update this data during membership, predecessor, or successor queries.

Instead of keeping just $\texttt{count}[x]$ (for duplicates) or a boolean membership indicator, we maintain a structure $\texttt{data}[x]$.

Insert

When we insert a new key x into the vEB tree, we also store the pointer (or the actual record) for its satellite data:

$$\texttt{data}[x] \leftarrow (\text{new satellite record for } x).$$

```
-------------------------------------------------------------------------------------
function VEB-INSERT(V, x, satellite_info):
    if V.min == NIL:
        // Tree empty: use vEB-Empty-Tree-Insert
        vEB-Empty-Tree-Insert(V, x)
        V.data[x] = satellite_info
        return
    if x < V.min:
        // If the new key is smaller than current min, swap
        swap x with V.min
        swap satellite_info with V.data[V.min]

    if V.u > 2:
        high_x = HIGH(x, V)
        low_x  = LOW(x, V)
        if V.cluster[high_x].min == NIL:
            VEB-INSERT(V.summary, high_x, /*some placeholder data*/)
            vEB-Empty-Tree-Insert(V.cluster[high_x], low_x)
            V.cluster[high_x].data[low_x] = satellite_info
```

```
        else:
            VEB-INSERT(V.cluster[high_x], low_x, satellite_info)

    if x > V.max:
        V.max = x
    V.data[x] = satellite_info
```

Delete

When deleting a key x, we remove its satellite data as well (`data`$[x]$ to `NIL`). If duplicates are involved, only remove the satellite data if the last occurrence of x is removed.

```
function VEB-DELETE(V, x):
    // standard vEB-Delete logic...
    // when removing x from the tree completely, also remove its satellite data:
    if (only occurrence of x):
        V.data[x] = NIL
```

Member, Predecessor, Successor

Membership queries can return the associated satellite data (if any) by simply returning `data`$[x]$ when `count`$[x] > 0$. The same can be applied for Predecessor and Successor.

The tree's structural operations remain the same, and each key-based operation takes the usual $O(\lg \lg U)$ time, as it stores the keys in a deterministic way using `High` and `Low`.

1.3 Exercise 20.3-3

Write pseudocode for a procedure that creates an empty van Emde Boas tree.

Answer:
We have two cases; one where the universe size $u = 2$, which is our base case, and where the universe size $u > 2$, which is our recursive case. Thus, u has to be a power of 2. We assume that u is never strictly less than 2, i.e. $u \geq 2$.
We first create a new vEB tree and initialize its min and max values to NIL, as they will both always be NIL regardless of the case.
For the base case where $u = 2$, then we simply create a cluster array of length 2 where both its elements are set to 0.
For the recursive case, we create a new empty vEB tree, summary, where its universe size $u = 2^{\lceil \lg(u)/2 \rceil}$, since u is a power of 2, and a new cluster array with the same size of $2^{\lceil \lg(u)/2 \rceil}$. We then run through the entire cluster array and for each entry, we set it to an empty vEB tree of universe size $u = 2^{\lceil \lg(u)/2 \rceil}$.
Finally, in both cases, the empty vEB tree is returned.

```
vEB-Tree-Create(u)
1   Let vEB be a new object
2   vEB.u = u       // u is a power of 2
3   vEB.min = NIL
4   vEB.max = NIL   // min and max are always null
5   if u == 2       // base case for u = 2
6       vEB.cluster = A[0..1]
7       vEB.cluster[0] = 0
8       vEB.cluster[1] = 0
9   else   // recursive case for u > 2
```

```
10      vEB.summary = vEB-Tree-Create(2[lg(u)/2])
11      vEB.cluster = new array A of size 2[lg(u)/2]
12      for i = 0 to 2[lg(u)/2]
13          vEB.cluster[i] = vEB-Tree-Create(2[lg(u)/2])
14  return vEB
```

1.4 Exercise 20.3-4

What happens if you call VEB-TREE-INSERT with an element that is already in the vEB tree? What happens if you call VEB-TREE-DELETE with an element that is not in the vEB tree? Explain why the procedures exhibit the behavior that they do. Show how to modify vEB trees and their operations so that we can check in constant time whether an element is present.
For the last part, remember that we already use $\Theta(u)$ space on the vEB tree. Thus, we can afford to maintain an auxiliary structure using $O(u)$ space without increasing the asymptotic space usage.

Answer:

Insert:
If inserting an already existing element x, all of the if-cases will be skipped, and the else-case on line 9 will always happen on insertion, and then it will recursively update until we reach the base case. If x is part of the base case, then nothing will be changed. This is because the vEB tree only stores info regarding the min and max values, as well as recursively update subsets in the tree, so it cannot tell if an element is already existing during insertion until the subsets have been updated recursively.

Deletion:
If deleting a non-existing element x, the function will be unable to find x, but still in all cases delete an element in the vEB tree regardless. If we look at the case on line 1; a tree with only one element, then the element in the tree will be deleted regardless of it existing or not. If we then look at the case on line 4, which requires that $u = 2$ and x cannot be equal 0 or 1, then the max element in the tree will always be deleted, regardless of what the element is. The cases on lines 9 and 21 cannot happen, since x does not exist in the vEB tree, so we cannot check if it is equal the min or max of the tree, so the max element will always still be deleted if x does not exist. This happens because, again, the tree does not check if x is present or not, and thus will continue to behave as if x is present.

Update Insert/Delete:
In both cases, to correct this behavior, we can create an auxiliary structure which checks if x is present or not. This is represented by an array of size u and indicates for each entry i if it is present or not, which can be represented through binary values, so if $[i] = 0$ then it is not present and if $[i] = 1$ then it is present. We can simply call this array `exists`.

For insertion, before we insert x, we check if $exists[x] == 1$, and if it is, then we skip the insertion to avoid inserting duplicate elements. If not, then we proceed insertion as normal.
For deletion, before we try to delete x, we run the same check for if $exists[x] == 0$, and if it is, then we skip deletion to avoid deleting non-existent elements. If not, then we proceed deletion as normal.

```
vEB-TREE-INSERT(V,x)
//new code for updated insertion
if exists[x] == 1:
    return      // nothing happens since x already exists
else
    // proceed as normal
```

```
vEB-TREE-DELETE(V,x)
//new code for updated deletion
```

```
if exists[x] == 0:
    return      // nothing happens since x doesn't exist so we can't delete it
else
    // proceed as normal
```

In both cases, the auxiliary space requires $O(u)$ space, as the array has size u, which is the universe size, and takes $O(1)$ time to perform, which does not affect the asymptotic space $\Theta(n)$ of the vEB-tree.

1.5 Exercise 20.3-5

Suppose that instead of $\sqrt{u}$ clusters, each with universe size $\sqrt{u}$, we constructed vEB trees to have $u^{1/k}$ clusters, each with universe size $u^{1-1/k}$, where $k > 1$ is a constant. If we were to modify the operations appropriately, what would be their running times? For the purpose of analysis, assume that $u^{1/k}$ and $u^{1-1/k}$ are always integers. (You may assume $k \geq 2$. Your solution should include the dependency on k.)

Answer:
With $u^{1/k}$ clusters, each of size $u^{1-1/k}$ (for constant $k \geq 2$), a single vEB operation involves:

$$T(u) \;=\; T\big(u^{1-1/k}\big) \;+\; T\big(u^{1/k}\big) \;+\; O(1).$$

Set $u = 2^m$ and define $S(m) = T(2^m)$. Then,

$$S(m) \;=\; S\big(m(1-1/k)\big) \;+\; S\big(m/k\big) \;+\; O(1).$$

For $k \geq 2$, this recurrence solves to $S(m) = O(\log m)$, implying

$$T(u) \;=\; O(\log \log u).$$

Thus, the running time per operation remains $O(\log \log u)$.

1.6 Exercise 20.3-6

Creating a vEB tree with universe size u requires $O(u)$ time. Suppose we wish to explicitly account for that time. What is the smallest number of operations n for which the amortized time of each operation in a vEB tree is $O(\lg \lg u)$?

Answer:
Initializing a vEB tree of universe size u takes $O(u)$ time. Suppose we perform n operations, each in $O(\lg \lg u)$. We want the one-time $O(u)$ cost to be amortized to $O(\lg \lg u)$ per operation.

Set

$$n \;=\; \frac{u}{\lg \lg u}.$$

Then the total time is

$$O(u) + n \cdot O(\lg \lg u) \;=\; O\big(u\big) + O\Big(\frac{u}{\lg \lg u} \cdot \lg \lg u\Big) \;=\; O(u).$$

Dividing by n yields an amortized $O(\lg \lg u)$ time per operation. Hence,

$$n \;\geq\; \frac{u}{\lg \lg u}.$$

Therefore, to hide the $O(u)$ initialization cost in an amortized $O(\lg \lg u)$-per-operation sense, we need at least

$$n \;=\; \Omega\Big(\frac{u}{\lg \lg u}\Big)$$

operations.

2 Exercises for Exact and FPT Algorithms

2.1 Exercise 1

What is the space needed for the $O^*(2^n)$ time TSP algorithm in Fomin-Kratsch?

Answer:
The space needed for the TSP algorithm in Fomin-Kratsch is polynomial. The algorithm uses dynamic programming with state encoding based on subsets of vertices and their tours.

The algorithm uses the Held Karp approach which is a dynamic programming table that computes the minimum costs of visiting a subset of vertices $S \subseteq V$ and ending in a vertex $v \in S$, where the states are updated iteratively.

Implementing this approach straightforward would require a table with $2^n \times n$ entries which takes exponential space. Fomin and Kratsch however optimizes this space complexity by recomputing intermediate results when necessary instead of storing all states. This optimization reduces the space requirements to polynomial. The result is a time complexity of $O^*(2^n)$ with a space complexity of polynomial.

2.2 Exercise 2

In the lecture, it was claimed that there was a mistake in the Fomin-Kratsch analysis of the $O^*(3^{n/3})$ time independent set algorithm. This exercise is about fixing the analysis.

(A) Complete the inductive step for $n \geq 2$ and $s = 1$.

(B) Use the given lemma to complete the inductive step for $n \geq 2$ and $s > 1$.

Answer:
...

2.3 Exercise 3

What is the space needed for the above recursive independent set algorithm from Fomin-Kratsch?

Answer:
For each recursive step in the algorithm, we choose a vertex v of minimum degree and branches on $v + 1$ subproblems. The space required for the entire recursive process is at worst $O(n)$, where n is the amount of vertices in the graph, since the depth of the recursion corresponds to the height of the graph G. The $v + 1$ subproblems are computed sequentially, so there is no need to allocate space for them.
On each level of the recursion, we need to store both the current graph representation (remaining subgraph) and information regarding the vertices and independent sets. If we assume this is represented by adjacency lists, the graph requires $O(m + n)$ space where m is the number of edges and n is the number of vertices in the graph.

With all this, we then have a total space requirement of:

$$O(n) \times O(m + n)$$

which is shortened to

$$O(n \times (m + n))$$

Thus, the space needed for the Exact MIS Branching algorithm is $O(n \times (m + n))$, where n is the amount of vertices and m is the amount of edges in the graph G.

2.4 Exercise 4

From the book on parameterized algorithms, we covered the "Bar Fight Prevention" problem (also called vertex cover): Given a graph $G = (V, E)$ with $n = |V|$ and $m = |E|$, find a vertex set $C \subseteq V$, $|C| \leq k$ such that all edges have an endpoint in C; or answer that no such C exists.

(a) Give an upper bound on the running time of this algorithm if you take into account the time it takes to identify U and the time it takes to check that a given $C \subseteq U$ is indeed a vertex cover of G.

(b) How good a running time can you get if you combine all the ideas, and perhaps some of your own? For example, $O(m + 2^k k^2)$ or, more challenging, $O(m + 2^k)$.

Answer:

(a) Upper Bound on Running Time

A common parameterized algorithm for vertex cover checks, for each edge, whether to include one endpoint or the other. This yields a worst-case bound of $O(2^k \cdot \text{poly}(n))$. However:

- Let U be the *universe* of vertices possibly in the cover (based on branching or reduction rules). Identifying U would take $O(m)$ or $O(n + m)$ time (scanning edges).
- For each candidate $C \subseteq U$, verifying if C is a vertex cover takes $O(m)$ time (we can check each edge to see if it's covered).

This means that, if we have 2^k subsets to test and each check is $O(m)$, the naive bound becomes:

$$O\big(m + 2^k \cdot m\big) \;=\; O(m + 2^k m).$$

(b) Improved Running Times

We aim for running times of either

$$O(m + n + 2^k k^2) \quad \text{or (if possible)} \quad O(m + n + 2^k).$$

We can use the subroutine BFP-Kernel(k, G) from the slides, which runs in $O(m + n)$, to reduce the graph to a kernel of size at most αk for some constant α. Then:

1. Kernelization: Apply BFP-Kernel(k, G) in $O(m + n)$. We obtain a smaller (kernel) graph G' with $\leq \alpha k$ vertices.
2. Branch/Check on Kernel: We can then try all subsets of G' up to size k. In the worst case, this gives us $O(2^k)$ subsets to check. If verifying each subset involves adjacency checks, we may get an extra factor of k^2, giving $O(2^k k^2)$.

Hence the total time is

$$O(m + n) \;+\; O\big(2^k k^2)\big),$$

leading to

$$O(m + n + 2^k k^2)$$