ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN NHÓM 3

Đề tài: van Emde Boas Tree

Môn học: Cấu trúc dữ liệu và giải thuật

Sinh viên thực hiện:

Nguyễn Đức Anh Khôi - 24120076

Nguyễn Quốc Nam - 24120098

Phan Tấn Vượng - 24120158

Nguyễn Quốc Bảo - 24120265

Giáo viên hướng dẫn:

GV. Lê Nhựt Nam

Mục lục

1 Giới thiệu

1.1 Bối cảnh

Trong lĩnh vực Khoa học Máy tính, để việc lưu trữ và truy xuất dữ liệu luôn được thực thi một cách hiệu quả nhất, việc nghiên cứu những cấu trúc dữ liệu là vô cùng quan trọng. Các cấu trúc dữ liệu truyền thống như mảng, danh sách liên kết và đặc biệt là cây tìm kiếm nhị phân đã đóng một vai trò quan trọng trong việc giải quyết vấn đề này. Tuy nhiên, đối với bài toán tìm successor (tìm phần tử kế tiếp) và predecessor (tìm phần tử trước) đối với một tập vũ trụ không đổi, các cấu trúc dữ liệu được nhắc đến bên trên vẫn bộc lộ nhiều hạn chế. Trong bối cảnh đó, cây van Emde Boas ra đời như là một giải pháp đột phá, cung cấp hiệu suất vượt trội và cải thiện những hạn chế của các cấu trúc dữ liệu truyền thống.

1.2 Lịch sử

Cây van Emde Boas được giới thiệu lần đầu tiên bởi Peter van Emde Boas vào năm 1977. Cấu trúc dữ liệu này ra đời từ nhu cầu cải thiện các thuật toán tìm kiếm, chèn, xóa, tìm successor và predecessor để đạt được độ phức tạp thời gian O(loglogM) trên một vũ trụ có số phần tử xác định. Cây vEB ra đời như một giải pháp đột phá, mở đường cho những biến thể của nó và góp phần quan trọng trong lĩnh vực nghiên cứu cấu trúc dữ liệu.

1.3 Giới hạn của báo cáo

Báo cáo này chỉ tập trung trình bày cấu trúc dữ liệu van Emde Boas Tree (vEB Tree) dưới góc nhìn lý thuyết và thực nghiệm cơ bản. Mặc dù đã đề cập đến một số biến thể và cấu trúc tương tự như Y-Fast Trie, Hierarchical Hash Table và các phương pháp thay thế khác, báo cáo chưa đi sâu vào phân tích chi tiết hoặc cài đặt của những biến thể này. Ngoài ra, phạm vi thử nghiệm hiệu năng cũng giới hạn ở quy mô dữ liệu nhỏ đến trung bình, chưa phản ánh đầy đủ hành vi ở các hệ thống thực tế lớn. Các yếu tố như tối ưu bộ nhớ, ứng dụng cụ thể trong hệ thống thực tế hoặc tích hợp với kiến trúc phần cứng cũng nằm ngoài phạm vi của báo cáo này.

1.4 Cấu trúc của báo cáo

Báo cáo được chia thành các phần chính như sau:

- **Phần 1 - Giới thiệu:** Trình bày một cảnh tổng quan về bối cảnh và lịch sử hình thành nên cấu trúc dữ liệu. Đồng thời cung cấp một vài thông tin liên quan đến bài báo cáo này.
- **Phần 2 - Chi tiết cấu trúc:** Miêu tả cấu trúc và cách hoạt động của cây vEB. Cung cấp mã giả và một phân tích độ phức tạp của cấu trúc.
- **Phần 3 - Kết quả thực nghiệm:** Cách tiến hành và kết quả của các thí nghiệm đánh giá hiệu quả của cây vEB.

- **Phần 4 - Kết luận:** Tổng kết những kết quả đã đạt được, rút ra nhận xét về ưu nhược điểm của cấu trúc cây vEB.
- **Phần 5 - Tài liệu tham khảo:** Liệt kê các nguồn tài liệu, bài viết, sách, video hoặc trang web đã được sử dụng để xây dựng nền tảng lý thuyết và triển khai nội dung báo cáo.

1.5 Bảng phân công

MSSV	Họ và tên	Nhiệm vụ phân công
24120076	Nguyễn Đức Anh Khôi	Viết báo cáo, soạn nội dung phần giới thiệu, tổng kết và vài nội dung trong phần thuật toán.
24120098	Nguyễn Quốc Nam	Soạn nội dung cho phần 2; Code của các biến thể vEB + hàm benchmark; file dữ liệu để test
24120158	Phan Tấn Vượng	Soạn nội dung phần 3 và 4 của bài báo cáo.
24120265	Nguyễn Quốc Bảo	Viết code cho file build; Tìm tài liệu và hỗ trợ nhóm

2 Một số kiến thức nền tảng

2.1 Từ điển có thứ tự (Ordered Dictionaries)

- Ordered Dictionary là một cấu trúc dữ liệu quản lý một tập hợp $S \subseteq U$ gồm các phần tử có thứ tự. Ví dụ: BST, AVL tree, vEB tree, x-fast trie, y-fast trie, Hierarchical Hash Table, ...
- Hỗ trợ các thao tác cơ bản sau:
  - insert(x): thêm phần tử x.
  - remove(x): xóa phần tử x khỏi S.
  - isMember(x): kiểm trả x có thuộc S không.
  - min(): trả về phần tử nhỏ nhất trong S.
  - max(): trả về phần tử lớn nhất trong S.
  - isEmpty(): kiểm tra S có rỗng không.
  - successor(x): trả về phần tử nhỏ nhất lớn hơn x.
  - predecessor(x): trả về phần tử lớn nhất nhỏ hơn x.

2.2 Đặc điểm dữ liệu được dùng

- Dữ liệu được dùng là các số nguyên nằm trong vũ trụ rời rạc hữu hạn U, nằm trong khoảng $[0, u-1]$ với u đủ nhỏ để nằm trong 1 machine word – thường là 32-bit hoặc 64-bit và bằng lũy thừa của 2.
- Ví dụ:
  - Nếu u = 2^{16} = 65,536
  - Thì $x \in [0, 65535]$ như 23213, 123, 65535, v.v.

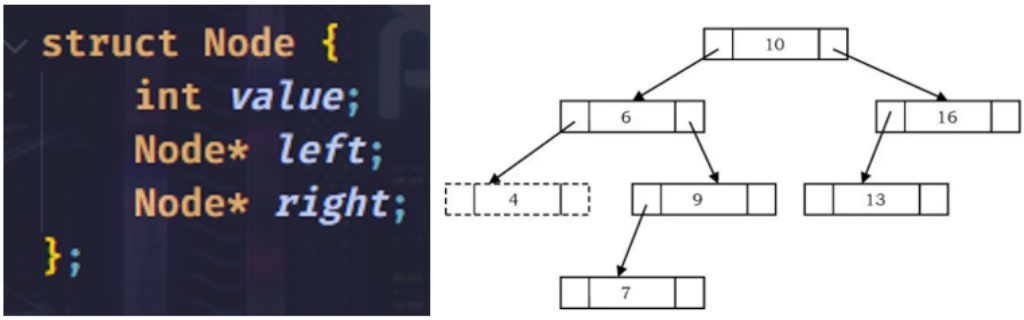
2.3 Mảng Bit (Bitvectors)

- Mảng bit là một chuỗi các bit (0,1) dùng để biểu diễn sự tồn tại của các phần tử.
- Bit tại vị trí i có giá trị bằng 1 nghĩa là i đang có trong tập và ngược lại.
- Các thao tác như tìm kiếm, chèn, xóa thực hiện nhanh bằng toán tử bit – bitwise operator $(<<, >>, \&, |, \wedge, \sim, =)$.
  - **isMember(x):** if((bitvector >> x) & 1)
  - **insert(x)**: bitvector |= (1 << x)
  - **remove(x):** bitvector & = ∼(1 << x)
- Mảng bit phân cấp (Tiered Bitvector):

- Được thiết kế để giảm độ phức tạp tìm kiếm cho successor() và predecessor().
- Ví dụ: với vũ trụ u = 216, ta chia thành:
  - Cấp 1(trên): Bitvector có kích thước 28 cho biết cụm bit nào đang có phần tử.
  - Cấp 0(dưới): Gồm 28 bit con mỗi cụm, lưu chi tiết các giá trị bit.

2.4 Đệ quy (Recrusion)

- Là phương pháp dùng trong các chương trình máy tính trong đó có một hàm tự gọi chính nó.
- Cấu trúc đệ quy xuất hiện khi một cấu trúc lồng ghép chính nó bên trong như một phần của định nghĩa.
- Thường dùng để giải quyết các bài toán bằng các chia nhỏ thành các trường hợp con nhỏ hơn.



Hình 1: Ví dụ về cấu trúc đệ quy Node với hai con trỏ `left` và `right` là các con trỏ Node*.

2.5 Cấu trúc của vEB Tree

VanEmdeBoasTree
universe_size, minimum, maximum, cluster_size: int
summary: VanEmdeBoasTree*
clusters: vector<VanEmdeBoasTree*>
VanEmdeBoasTree(size: int): constructor
~VanEmdeBoasTree(): destructor
min(), max(): int
isMember(x: int), isEmpty(): bool
insert(x: int), remove(x: int): void
successor(x: int), predecessor(x: int): int
hight(x: int), low(x: int), generateIndex(x: int, y: int): int
emptyInsert(x: int): void

Hình 2: Câu trúc của cây vEB

- **universe_size**: kích thước vũ trụ u, là phạm vi giá trị mà cây hỗ trợ (lũy thừa của 2).

- **minimum**: giá trị nhỏ nhất hiện có trong cây.
- **maximum**: giá trị lớn nhất hiện có trong cây.
- **cluster_size**: kích thước của mỗi cụm con ($\sqrt{u}$), dùng để chia nhỏ vũ trụ.
- **summary**: cây vEB tóm tắt, theo dõi cụm nào có phần tử.
- **clusters**: vector các cây vEB con, mỗi cụm quản lý một phần vũ trụ u.
- **VanEmdeBoasTree()**: hàm tạo, khởi tạo cây với kích thước size = u.
- **VanEmdeBoasTree()**: hàm hủy, xóa đệ quy các cụm con và cây summary.
- **min()**, **max()**, **isEmpty()**, **isMember()**, **insert()**, **remove()**, **successor()**, **predecessor()**: là các phương thức tương tự như của cấu trúc từ điển có thứ tự kiểu số nguyên.
- **high()**: trả về chỉ số cụm chứa x, dùng để truy cập clusters.
- **low()**: trả về vị trí của x trong cụm của nó.
- **generateIndex()**: gộp chỉ số cụm x và vị trí y thành chỉ số đầy đủ.
- **emptyInsert()**: dùng khi insert vào cây đang rỗng, gán min = max = x.

2.6 Các biến thể và thay thế của vEB tree

- Các biến thể của cây vEB tree được tạo ra để cải tiến bộ nhớ, độ phức tạp phù hợp với từng bài toán cụ thể. Gồm có: **X-Fast Trie**, **Y-Fast Trie**, **Bitvector vEB**,…
- Ngoài ra còn có các cấu trúc dữ liệu khác dùng để giải quyết bài toán **Integer Ordered Dictionary** ngoài các vEB tree như: **BST**, **AVL tree**, **Red-Black tree**, **Sorted Hash Map**, **Hierarchical Hash Table**, **Direct Mapped Cache**,…
- Đặc biệt với cấu trúc **Y-Fast Trie** (bản tối ưu bộ nhớ của X-Fast Trie – một biến thể của vEB bản gốc) và **Hierarchical Hash Table** (đệ quy phân tầng tương tự như vEB bản gốc).
  - **Y-Fast Trie**: Sử dụng X-Fast Trie làm summary và mỗi cụm con là balanced BST (AVL Tree, Red-Black Tree,…). Vì thế tối ưu hơn X-Fast Trie về bộ nhớ và chạy insert, delete, search trong $O(\log\log u)$.
  - **Hierarchical Hash Table**: sử dụng phương pháp phân tầng giống của vEB nhưng không cần chia cụm đều như vEB. Tuy nhiên insert, delete chậm vì phải cập nhật lại nhiều bảng con.

3 Chi tiết cấu trúc Van Emde Boas Tree (VEB Tree)

3.1 Cấu trúc VEB Tree

Van Emde Boas Tree (VEB Tree) là một cấu trúc dữ liệu dùng để lưu trữ và thao tác trên một tập hợp số nguyên trong phạm vi $[0, u-1]$, với $u = 2^k$ (universe size). Cấu trúc này cho phép các thao tác trên từ điển (insert, delete, search) với độ phức tạp $O(\log \log u)$, nhanh hơn nhiều so với các cấu trúc dữ liệu thông thường như Binary Search Tree ($O(\log n)$) hoặc Hash Table ($O(1)$ trung bình nhưng không hỗ trợ thao tác thứ tự (VD: min, max, predecessor, successor, duyệt các phần tử theo thứ tự tăng dần/giảm dần)).

3.1.1 Các thành phần chính

- **minimum và maximum:**
  - Lưu giá trị nhỏ nhất và lớn nhất trong cây.
  - Nếu cây rỗng, minimum = maximum = -1.
  - Giúp tối ưu các thao tác min(), max(), successor(), predecessor().
- **summary:**
  - Một VEB Tree con có kích thước $\sqrt{u}$, dùng để theo dõi các cụm (clusters) không rỗng.
  - Nếu một cụm chứa phần tử, bit tương ứng trong summary được bật.
- **clusters:**
  - Mảng gồm $\sqrt{u}$ VEB Tree con, mỗi cụm quản lý một phạm vi con có kích thước $\sqrt{u}$.
  - Một số x được phân tách thành high(x) (chỉ số cụm) và low(x) (giá trị trong cụm).

3.1.2 Phân tách đệ quy (Recursive Decomposition)

- VEB Tree sử dụng phép phân tách đệ quy theo hàm mũ:
  - $u = 2^{(2k)} \Rightarrow \sqrt{u} = 2^k$
  - Mỗi tầng đệ quy giảm kích thước vấn đề từ u xuống $\sqrt{u}$, dẫn đến độ phức tạp $O(\log \log u)$.

3.2 Giải thuật (Pseudo Code)

3.2.1 Thao tác insert(x)

- Nếu cây rỗng, gán min = max = x.
- Nếu $x <$ min, hoán đổi x và min, sau đó chèn giá trị mới.
- Nếu $u > 2$, chèn x vào cụm tương ứng và cập nhật summary:
  - Tìm cluster $h =$ high(x), và vị trí trong cluster $l =$ low(x).

  - Nếu cluster chưa có gì → insert vào summary, rồi khởi tạo cluster con.
  - Ngược lại → đệ quy chèn vào cluster.
- Cập nhật maximum nếu cần.

```
Function Insert(x):
    If tree is empty:
        min ← x
        max ← x
    Else:
        If x < min:
            Swap(x, min)
        If universe_size > 2:
            h ← high(x)
            l ← low(x)
            If clusters[h] is null:
                clusters[h] ← new VEBTree(cluster_size)
            If clusters[h] is empty:
                summary.Insert(h)
            clusters[h].Insert(l)
        If x > max:
            max ← x
End Function
```

3.2.2 Thao tác remove(x)

- Nếu chỉ có 1 phần tử, xóa và đặt min = max = -1.
- Nếu cây có 2 phần tử thì xóa 1 phần tử, phần tử còn lại là min và max.
- Nếu x == min, tìm min mới từ summary và các cụm con.
- Đệ quy để xóa x khỏi cụm tương ứng, cập nhật summary nếu cụm trở thành rỗng.

```
Function Remove(x):
    If min == max:
        min ← -1
        max ← -1
    Else if universe_size == 2:
        If x == 0:
            min ← 1
            max ← 1
        Else:
```

```
            min ← 0
            max ← 0
    Else:
        If x == min:
            first_cluster ← summary.min()
            x ← generateIndex(first_cluster, clusters[first_cluster].min
                ())
            min ← x
        h ← high(x)
        l ← low(x)
        clusters[h].Remove(l)
        If clusters[h] is empty:
            summary.Remove(h)
        If x == max:
            If summary is empty:
                max ← min
            Else:
                max_cluster ← summary.max()
                max ← generateIndex(max_cluster, clusters[max_cluster].
                    max())
End Function
```

3.2.3 Thao tác successor: tìm phần tử nhỏ nhất trong tập hợp lớn hơn x (kế tiếp)

- Nếu x nhỏ hơn minimum, successor(x) chính là minimum.
- Nếu x lớn hơn hoặc bằng maximum, không có phần tử kế tiếp → trả về -1.
- Nếu x nằm trong một cụm (cluster) có phần tử lớn hơn x, thì gọi đệ quy tìm successor trong cụm đó.
- Nếu cụm của x không có successor, ta gọi summary->successor(cluster_index) để tìm cụm kế tiếp có phần tử.
- Khi có cụm kế tiếp, lấy phần tử nhỏ nhất trong cụm đó làm successor.

```
Function Successor(x):
    If universe_size == 2:
        If x == 0 AND max == 1:
            Return 1
        Return -1
    If min != -1 AND x < min:
        Return min
```

```
    h ← high(x)
    l ← low(x)
    maxLow ← clusters[h].max()
    If clusters[h] contains element > l (maxLow   -1 AND l < maxLow):
        offset ← clusters[h].Successor(l)
        Return generateIndex(h, offset)
    succCluster ← summary.Successor(h)
    If succCluster == -1:
        Return -1
    offset ← clusters[succCluster].min()
    Return generateIndex(succCluster, offset)
End Function
```

3.2.4 Thao tác predecessor: tìm phần tử lớn nhất trong tập hợp nhỏ hơn x (liền trước)

- Nếu x lớn hơn maximum, predecessor(x) chính là maximum.
- Nếu x nhỏ hơn hoặc bằng minimum, không có phần tử liền trước $\rightarrow$ trả về -1.
- Nếu x nằm trong một cụm có phần tử nhỏ hơn x, thì gọi đệ quy tìm predecessor trong cụm đó.
- Nếu cụm không có predecessor, ta gọi summary->predecessor(cluster_index) để tìm cụm trước đó có phần tử.

```
Function Predecessor(x):
    If universe_size == 2:
        If x == 1 AND min == 0:
            Return 0
        Return -1
    If max != -1 AND x > max:
        Return max
    h ← high(x)
    l ← low(x)
    If clusters[h] contains element < l:
        offset ← clusters[h].Predecessor(l)
        Return generateIndex(h, offset)
    predCluster ← summary.Predecessor(h)
    If predCluster == -1:
        If x > min: Return min
        Else: Return -1
    offset ← clusters[predCluster].max()
    Return generateIndex(predCluster, offset)
```

```
End Function
```

3.2.5 Thao tác kiểm tra thành viên

- Kiểm tra nếu $x ==$ min hoặc $x ==$ max $\rightarrow$ trả về true ngay ($O(1)$)
- Nếu cây chỉ chứa 2 phần tử (universe_size ≤ 2) $\rightarrow$ trả về false
- Chia x thành:
  - cluster = high(x) (chỉ số cụm)
  - value = low(x) (giá trị trong cụm)
- Nếu cụm đó chưa tồn tại $\rightarrow$ false
- Đệ quy kiểm tra value trong cụm con

```
Function isMember(x):
    If x == min OR x == max:
        Return True
    If universe_size <= 2:
        Return False
    h ← high(x)
    l ← low(x)
    If clusters[h] is null:
        Return False
    Return clusters[h].isMember(l)
End Function
```

3.2.6 Thao tác khởi tạo

- Thiết lập kích thước vũ trụ (**universe_size**)
- Khởi tạo min = max = -1 (cây rỗng)
- Trường hợp cơ sở (size ≤ 2): nếu kích thước bé hơn hoặc bằng 2 ($u \leq 2$) thì không cần phân rã ra nữa $\rightarrow$ không cần cụm con
- Trường hợp tổng quát (size > 2):
  - Chia kích thước thành $\sqrt{u}$ clusters
  - Tạo summary để theo dõi clusters nào có phần tử
  - Khởi tạo clusters là mảng chứa các cây vEB con kích thước nhỏ hơn

```
Function VEBTree(size):
    universe_size ← size
```

```
    min ← -1
    max ← -1
    summary ← null
    If size > 2:
        cluster_size ←√ size
        clusters ← new array of size cluster_size initialized with null
summary ← new VEBTree(cluster_size)
    Else:
        cluster_size ← 0
        clusters ← empty array
End Function
```

3.2.7 Các hàm hỗ trợ

1. **high(x):**
   - Trả về chỉ số cụm chứa x.
   - Ví dụ: $u = 16$, high(5) $= 5/4 = 1$ (vì $\sqrt{16} = 4$).

```
Function high(x):
    Return x ÷ cluster_size
End Function
```

2. **low(x):**
   - Trả về giá trị của x trong cụm hiện tại.
   - Ví dụ: $u = 16$, low(5) $= 5 \pmod 4 = 1$.

```
Function low(x):
    Return x mod cluster_size
End Function
```

3. **generateIndex(cluster, value):**
   - Kết hợp chỉ số cụm và giá trị để tạo ra giá trị ban đầu.
   - Ví dụ: generateIndex(1, 1) $= 1 * 4 + 1 = 5$.

```
Pseudocode:
Function generateIndex(cluster, value):
    Return cluster × cluster_size + value
End Function
```

Cải tiến hàm high(x) và low(x): Với những vũ trụ có U là một lũy thừa của 2, $U = 2^k$, mỗi số x trong U có biểu diễn nhị phân là một dãy bit với k bit. Khi đó:

- high(x) chính là nửa đầu dãy bit biểu diễn nhị phân của x.
- low(x) chính là nửa sau dãy bit biểu diễn nhị phân của x.

Lợi dụng điều này ta có thể tăng tốc độ tính toán của high(x) và low(x) bằng bit manipulation thay cho việc dùng hàm `sqrt()` và phép chia modulo. Ví dụ: universe_size = $16 = 2^4$, $x = 5$. half = $\log_2(16)/2 = 4/2 = 2$.

- high(x) = $x >> 2 = 1$
  - Tính tay: $x = 5$, biểu diễn nhị phân của x là 0101_2.
  - high(x) = $5 >> 2$: dịch phải 5 2 bit: $0101_2 \to 0001_2$.
  - $\to$ high(x) = 1 (1 có biểu diễn nhị phân là 0001_2)
- low(x) = x & ((1 << half) − 1)
  - Tính tay: $x = 5$, biểu diễn nhị phân của x là 0101_2.
  - 1 << 2: dịch trái 1 2 bit: $0001_2 \to 0100_2$.
  - (1 << 2) − 1: $0100_2 - 0001_2 = 0011_2$.
  - low(x) = 5 & ((1 << half) − 1) = 0101_2 AND $0011_2 = 0001_2$.
  - $\to$ low(x) = 1 (1 có biểu diễn nhị phân là 0001_2).

Vậy high(5) = 1 và low(5) = 1. (tương tự như dùng `sqrt()` và modulo)

Lưu ý: Chỉ hoạt động khi **universe_size là một lũy thừa của 2**. Hiệu quả: Tăng đáng kể hiệu suất và tốc độ xử lí của cây vEB.

3.2.8 Độ phức tạp: $O(\log \log u)$ đối với cả insert, search, successor, isMember.

- **Ưu điểm:**
  - Tốc độ cực nhanh với u lớn (ví dụ: $u = 2^{32} \to \log \log u \approx 5$).
  - Hỗ trợ mọi thao tác có thứ tự một cách hiệu quả.
- **Nhược điểm:**
  - Tốn bộ nhớ do lưu trữ nhiều cây con.
  - Khó cài đặt hơn so với BST hoặc bảng băm.

3.3 Phân tích độ phức tạp

3.3.1 Thời gian

Dưới đây là độ phức tạp thời gian của các thao tác trên vEB:

Thao tác	**Độ phức tạp**
`insert(x)`	$O(\log \log u)$
`remove(x)`	$O(\log \log u)$
`isMember(x)`	$O(\log \log u)$
`min()`, `max()`	$O(1)$
`successor(x)`, `predecessor(x)`	$O(\log \log u)$

3.3.2 Không gian

- **Tổng bộ nhớ:** $O(u)$ (do lưu trữ đệ quy các cây con).
- **Tối ưu hóa:** Có thể giảm bộ nhớ bằng cách sử dụng hash table thay cho mảng clusters, nhưng điều này sẽ làm tăng độ phức tạp thời gian.

3.3.3 So sánh với các cấu trúc khác

Bảng dưới đây so sánh vEB Trê với một số cấu trúc dữ liệu phổ biến khác về độ phức tạp thời gian và không gian:

Cấu trúc	**insert/delete**	**search**	**successor/predecessor**	**Bộ nhớ**
VEB Tree	$O(\log \log u)$	$O(\log \log u)$	$O(\log \log u)$	$O(u)$
BST	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Hash Table	$O(1)$ (TB)	$O(1)$ (TB)	Không hỗ trợ	$O(n)$
Mảng địa chỉ trực tiếp	$O(1)/O(n)$	$O(n)$	$O(n)$	$O(n)$

Nhận xét:

- **Ưu điểm của VEB Tree:**
  - Các thao tác có độ phức tạp $O(\log \log u)$, rất nhanh với u (kích thước vũ trụ) lớn.
  - Hỗ trợ các thao tác thứ tự (`successor`, `predecessor`) hiệu quả.
- **Nhược điểm của VEB Tree:**
  - Tốn bộ nhớ ($O(u)$) do cấu trúc phân cấp.
  - Khó cài đặt hơn so với BST hoặc Hash Table.

4 Kết quả thực nghiệm

4.1 Cách cài đặt

4.1.1 Cấu trúc dữ liệu

Dưới đây là code C++ cho cấu trúc dữ liệu vEB Tree:

```
class VanEmdeBoasTree {
private:
    int universe_size;
    int minimum, maximum;
    VanEmdeBoasTree* summary
    vector<VanEmdeBoasTree*> clusters;
    int cluster_size;
    int high(int x) { return x / cluster_size; }
    int low(int x) { return x % cluster_size; }
    int generateIndex(int cluster, int value) {
        return cluster * cluster_size + value;
    }
public:
    VanEmdeBoasTree(int size);
    ~VanEmdeBoasTree();
    void insert(int x);
    void remove(int x);
    bool isMember(int x);
    int successor(int x);
    int predecessor(int x);
    int min() { return minimum; }
    int max() { return maximum; }
};
```

- **universe_size**: kích thước giá trị có thể có.
- **minimum, maximum**: giá trị phần tử nhỏ nhất và lớn nhất trong cây.
- **summary**: dùng để theo dõi cây con nào không rỗng (có ít nhất 1 phần tử).
- **clusters**: mảng cây vEB con, mỗi cái chứa một phần không gian.

4.1.2 Khởi tạo và hủy

Dưới đây là code C++ cho thao tác khởi tạo và hủy vEB Tree:

```
Van_Emde_Boas(int size) {
        universe_size = size;
        minimum = -1;
        maximum = -1;

        if (size <= 2) {
            summary = nullptr;
            clusters = vector<Van_Emde_Boas*>(0, nullptr);
        }
        else {
            int cluster_size = ceil(sqrt(size));
            summary = new Van_Emde_Boas(cluster_size);
            clusters = vector<Van_Emde_Boas*>(cluster_size, nullptr);
            for (int i = 0; i < cluster_size; i++) {
                clusters[i] = new Van_Emde_Boas(cluster_size);
            }
        }
    }

~VanEmdeBoasTree() {
    for (auto cluster : clusters) delete cluster;
    delete summary;
}
```

4.1.3 Tối ưu hoá

- **Dynamic resizing:** Tự động điều chỉnh kích thước khi u không phải lũy thừa 2.
- **Memory pooling:** Dùng memory pool để giảm cấp phát động.
- **Bit compression:** Nén dữ liệu cho các trường hợp u lớn.

5 Kết quả thực nghiệm

5.1 Đo lường hiệu năng thực tế

5.1.1 Thiết lập thử nghiệm

- **Môi trường thử nghiệm:**
  - CPU: Intel Core i7-11800H (8 cores, 2.3GHz)
  - RAM: 16GB DDR4

  - OS: Windows 11 Pro
  - VSCode: Version 1.82.2
  - Optimization: -O3

- **Dữ liệu:** Sinh ngẫu nhiên các tập từ 2^9 đến 2^{23} phần tử.
- **So sánh với:**
  - Các kiểu dữ liệu thay thế: BST, Hash Table, mảng địa chỉ trực tiếp.
  - Các biến thể của vEB Tree: Y-Fast Trie, Hierarchical Hash Table.

5.1.2 Thực nghiệm đo thời gian

vEB Tree Với các thao tác trên vEB Tree, kết quả về thời gian chạy như sau:

Size	**insert**	**remove**	**isMember**	**min**	**max**	**successor**	**predecessor**
2^9	0	0	0	0	0	0	0
2^{12}	1	1	1	0	0	1	1
2^{14}	6	5	3	0	0	4	4
2^{16}	26	28	17	0	0	19	16
2^{20}	562	330	614	0	0	242	261
2^{23}	4409	3756	5127	0	0	1644	3429

Bảng 1: Thời gian thực hiện các thao tác trên vEB Tree (đơn vị: ms)

BST Để so sánh, ta thử nghiệm và thu về kết quả của các thao tác trên cây BST như sau:

Size	**insert**	**remove**	**isMember**	**min**	**max**	**successor**	**predecessor**
2^9	0	0	0	0	0	0	0
2^{12}	3	1	3	2	3	4	4
2^{14}	17	10	21	15	13	25	28
2^{16}	46	16	28	40	43	65	57
2^{20}	592	474	774	615	589	876	844
2^{23}	9278	8541	7688	450	465	12860	13004

Bảng 2: Thời gian thực hiện các thao tác trên BST (đơn vị: ms)

Hash Table Để so sánh, ta thử nghiệm và thu về kết quả của các thao tác trên Hash Table như sau:

Size	insert	remove	isMember	min	max	successor	predecessor
2^9	0	0	0	N/A	N/A	N/A	N/A
2^{12}	2	0	0	N/A	N/A	N/A	N/A
2^{14}	16	6	6	N/A	N/A	N/A	N/A
2^{16}	15	8	5	N/A	N/A	N/A	N/A
2^{20}	306	185	133	N/A	N/A	N/A	N/A
2^{23}	1804	2200	2145	N/A	N/A	N/A	N/A

Bảng 3: Thời gian thực hiện các thao tác trên Hash Table (đơn vị: ms)

Mảng địa chỉ trực tiếp Để so sánh, ta thử nghiệm và thu về kết quả của các thao tác trên mảng địa chỉ trực tiếp như sau:

Size	insert	remove	isMember	min	max	successor	predecessor
2^9	0	0	0	5	6	6	4
2^{12}	0	0	0	51	45	48	42
2^{14}	4	3	6	180	176	168	179
2^{16}	5	4	6	3012	3142	3254	3268
2^{20}	12	21	18	40900	41235	39867	40598
2^{23}	301	365	334	6786960	6598756	6345889	6124086

Bảng 4: Thời gian thực hiện các thao tác trên Mảng địa chỉ trực tiếp (đơn vị: ms)

Các biến thể và cấu trúc tương tự vEB tree

Y-Fast Trie Kết quả về tốc độ các thao tác trên Y-Fast Trie:

Size	insert	remove	isMember	min	max	successor	predecessor
2^9	0	0	0	0	0	0	0
2^{12}	1	1	1	0	0	2	2
2^{14}	5	6	3	0	0	8	9
2^{16}	23	24	10	0	0	39	37
2^{20}	386	113	48	0	0	316	339
2^{23}	4355	507	312	0	0	2791	2881

Bảng 5: Thời gian thực hiện các thao tác trên Y-Fast Trie (đơn vị: ms)

Hierarchical Hash Table Kết quả về tốc độ các thao tác trên Hierarchical Hash Table:

Size	insert	remove	isMember	min	max	successor	predecessor
2^9	1	1	0	0	0	0	0
2^{12}	8	8	1	0	0	1	1
2^{14}	40	40	4	0	0	4	5
2^{16}	222	204	20	0	0	19	22
2^{20}	3196	2656	212	0	0	205	234
2^{23}	33199	27965	2692	0	0	2295	2125

Bảng 6: Thời gian thực hiện các thao tác trên Hierarchical Hash Table (đơn vị: ms)

5.2 So sánh

Van Emde Boas Tree:

1. **Thao tác cơ bản:**
   - `insert()` và `remove()`:
     - Với $u \leq 2^{16}$: Hiệu suất cực tốt ($< 30\mu s$).
     - Khi $u = 2^{20}$: Tăng đột biến ($\sim 300 - 600\mu s$) do chi phí phân tách đệ quy.
     - Tại $u = 2^{23}$: Đạt $\sim 4ms$ (phù hợp với lý thuyết $O(\log \log u)$).
   - `isMember()`:
     - Biến động lớn ($3\mu s$ ở $2^{14} \rightarrow 5127\mu s$ ở 2^{23}).
     - Nguyên nhân: Phụ thuộc vào độ sâu đệ quy và cache misses.
2. **Thao tác thứ tự: vượt trội khi** $u \geq 2^{16}$
   - `min()`/`max()`:
     - Luôn là $O(1)$ ($0\mu s$ trên mọi kích thước).
     - Tối ưu nhờ lưu trữ riêng biệt.
   - `successor()` và `predecessor()`:
     - Chênh lệch lớn giữa các kích thước.
     - Ví dụ: `successor()` mất $1644\mu s$ ở 2^{23} do phải duyệt qua nhiều tầng summary.
     - Ở 2^{23}: `successor` nhanh hơn BST $7.5x$ ($1,600\mu s$ vs $12,000\mu s$).

BST (AVL):

- Cân bằng giữa tốc độ và chức năng.
- Hiệu suất ổn định trên toàn dải.
- Chi phí cân bằng làm chậm `insert/delete`.
- `Successor` ổn định ở $0.5 - 3.8\mu s$ với $u < 2^{12}$.

Hash Table:

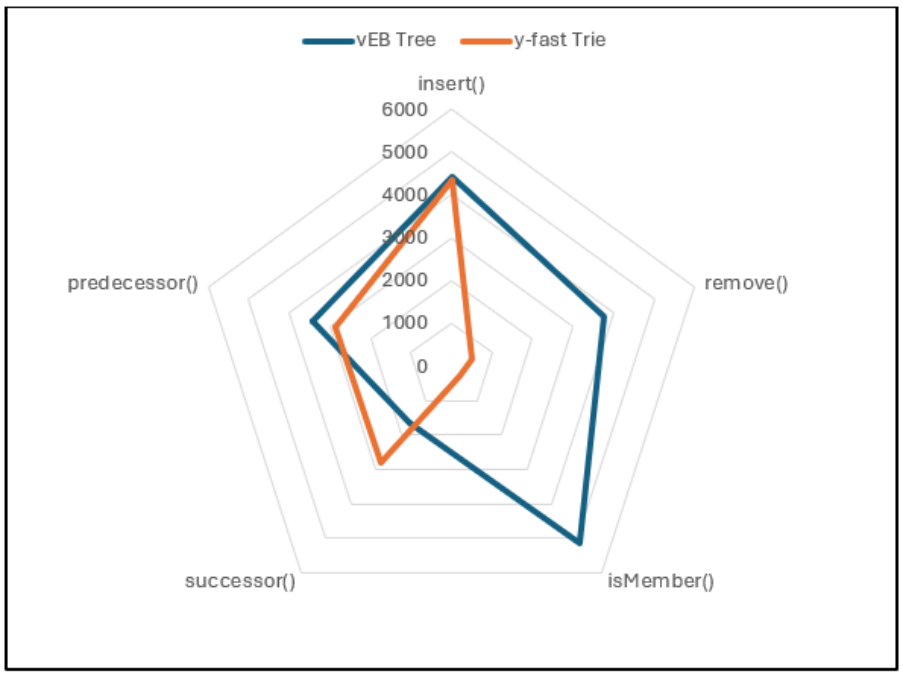
- Thao tác cơ bản nhanh nhất ($0.1\mu s$).
- Hiệu năng giảm khi load factor $> 70\%$.
- Tốc độ `insert/search` nhanh gấp $50 - 100x$ VEB ở 2^{23}.
- Không hỗ trợ thao tác thứ tự.
- Collision tăng khi $u > 2^{20}$.

Mảng địa chỉ trực tiếp:

- `Insert/Search` nhanh nhất ($0.05 - 0.1\mu s$), cực nhanh ở 2^{12} ($0.1\mu s$).
- Ordered operations chậm đặc biệt ở u cao (phải quét toàn bộ mảng).

Các biến thể và cấu trúc tương tự vEB tree

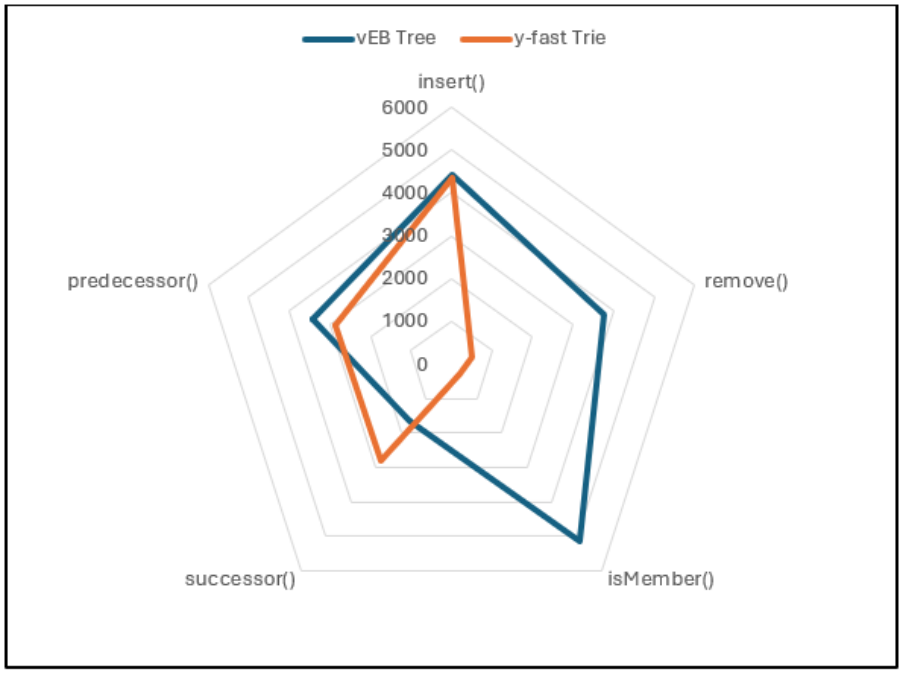
Y-Fast Trie Kết quả:



Biểu đồ 1: Thời gian chạy (ms) của các thao tác trên Y-Fast Trie

- Hiệu năng ổn định kể cả với dữ liệu rất lớn 2^{23}. Bộ nhớ sử dụng được tối ưu hơn vEB.
- Phù hợp với các phương thức tìm kiếm.
- Độ phức tạp: $O(\log \log u)$.

Hierarchical Hash Table Kết quả:



Biểu đồ 2: Thời gian chạy (ms) của các thao tác Hierarchical Hash Table

- Tương tự vEB về mặt ý tưởng chia tầng, nhưng hiệu năng thực tế tệ hơn do chi phí quản lý mỗi tầng quá cao.
- Thời gian cho `insert` và `remove` tăng gấp 10 lần thời gian so với vEB hoặc Y-Fast ở 2^{23} phần tử.
- Độ phức tạp: $O(\log u)$.

5.2.1 Kết luận

- $2^{12} – 2^{16}$ **(4K - 65K):**
  - Ưu tiên Hash Table nếu không cần ordered ops.
  - Dùng BST nếu cần `successor/predecessor`.
  - Tránh VEB Tree (overkill).
- $2^{16} – 2^{20}$ **(65K - 1M):**
  - VEB Tree trở nên hợp lý nếu cần ordered ops thường xuyên.
  - BST vẫn là lựa chọn an toàn.
  - Hash Table vẫn tốt nhất cho pure lookup (chỉ yêu cầu kiểm tra sự tồn tại hoặc lấy giá trị).
- $2^{20} – 2^{23}$ **(1M - 8M):**
  - VEB Tree chỉ dùng khi:
    - Có đủ RAM.
    - Yêu cầu `successor/min/max` cực nhanh.
  - BST phù hợp cho hệ thống general-purpose (có thể đáp ứng nhiều loại thao tác (lookup, insert, delete, ordered ops) mà không cần tối ưu cực đoan cho bất kỳ tác vụ cụ thể nào:
    - Cân bằng giữa tốc độ và bộ nhớ.
    - Hỗ trợ đa dạng thao tác ở mức hiệu suất chấp nhận được.
    - Dễ triển khai và bảo trì.
  - Hash Table cho hệ thống write-intensive (Hệ thống có tần suất ghi dữ liệu (insert/update/delete) cao hơn đáng kể so với đọc (read)).

6 Tổng kết

Cấu trúc dữ liệu **cây van Emde Boas (vEB Tree)** là một giải pháp đột phá để quản lý và thao tác trên tập hợp các số nguyên trong một vũ trụ hữu hạn và rời rạc, đặc biệt hiệu quả với các thao tác tìm kiếm phần tử kế tiếp (**successor**) và phần tử trước đó (**predecessor**).

6.1 Cấu trúc làm được gì

Cây vEB cung cấp hiệu suất vượt trội cho các thao tác từ điển cơ bản như **insert**, **remove**, **isMember**, **min**, **max**, **successor**, và **predecessor** với độ phức tạp thời gian $O(\log \log u)$, trong đó u là kích thước của vũ trụ (phạm vi giá trị có thể có). Điều này nhanh hơn đáng kể so với các cấu trúc dữ liệu truyền thống như cây tìm kiếm nhị phân cân bằng (BST) có độ phức tạp $O(\log n)$ (với n là số phần tử hiện có trong cây) hoặc Hash Table chỉ cung cấp $O(1)$ trung bình nhưng không hỗ trợ các thao tác thứ tự.

Cấu trúc của vEB Tree dựa trên nguyên tắc **phân tách đệ quy**, chia vũ trụ thành các cụm con nhỏ hơn và sử dụng một cây tóm tắt (**summary tree**) để theo dõi các cụm không rỗng. Các thành phần chính bao gồm:

- **minimum** và **maximum**: Lưu trữ giá trị nhỏ nhất và lớn nhất để tối ưu các thao tác `min()`, `max()`, `successor()`, `predecessor()`.
- **summary**: Một vEB Tree con theo dõi các cụm không rỗng.
- **clusters**: Mảng các vEB Tree con, mỗi cụm quản lý một phạm vi con của vũ trụ.

6.2 Ứng dụng

Với hiệu suất vượt trội trong các thao tác thứ tự, vEB Tree đặc biệt phù hợp cho các ứng dụng đòi hỏi:

- **Truy vấn thứ tự nhanh chóng**: Các hệ thống cần tìm phần tử kế tiếp hoặc trước đó của một giá trị cụ thể một cách hiệu quả, chẳng hạn như trong cơ sở dữ liệu thời gian thực hoặc các thuật toán lập lịch.
- **Quản lý tập hợp số nguyên lớn**: Khi kích thước vũ trụ u lớn (2^{16} đến 2^{23} hoặc hơn), vEB Tree cho thấy ưu thế rõ rệt so với BST và các cấu trúc khác trong các thao tác có thứ tự.
- **Hệ thống cần thao tác đồng thời trên cả việc thêm/xóa và truy vấn thứ tự**: vEB Tree cân bằng giữa tốc độ của các thao tác cơ bản và thao tác thứ tự, là lựa chọn tốt hơn so với Hash Table (không hỗ trợ thứ tự) và mảng địa chỉ trực tiếp (thao tác thứ tự chậm).

6.3 Cải thiện nếu có thêm thời gian

Nếu có thêm thời gian để cải thiện cấu trúc vEB Tree, những điểm sau đây có thể được xem xét:

- **Tối ưu bộ nhớ**: vEB Tree gốc có thể tiêu tốn nhiều bộ nhớ, đặc biệt khi vũ trụ u lớn nhưng số lượng phần tử thực tế lại ít
- **Tích hợp với kiến trúc phần cứng**: Tối ưu hóa cấu trúc để tận dụng cache của CPU tốt hơn, giảm thiểu cache misses, đặc biệt khi dữ liệu lớn, có thể cải thiện đáng kể hiệu suất thực tế.
- **Phân tích sâu hơn các biến thể**: Đi sâu vào phân tích và cài đặt các biến thể như Y-Fast Trie và Hierarchical Hash Table để hiểu rõ hơn về ưu nhược điểm của chúng trong các tình huống cụ thể và đề xuất hướng kết hợp hoặc cải tiến mới.