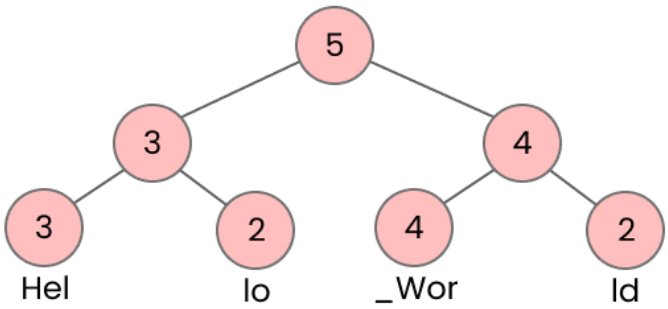
# Rope

rope หรือ cord เป็น data structure ที่ใช้ในการเก็บและจัดการ string ที่มีความยาวมาก ๆ เมื่อเทียบกับ string แบบปกติแล้ว rope จะสามารถทำการแทรกหรือลบ substring ได้เร็วกว่ามาก และนอกจากนั้น rope ยังสามารถเขียนให้มีความสามารถในการ undo หรือ redo การทำ operation ต่าง ๆ ได้อีกด้วย ทำให้ rope ถูกนำไปใช้ในโปรแกรม Text Editor การเขียนอีเมล และ string buffer ที่จัดการกับข้อความยาว ๆ

## การเก็บข้อมูล




rope จะทำการแบ่งเก็บ string ยาว ๆ เป็นหลาย ๆ substring โดยเก็บข้อมูลเป็น binary tree แต่ละ node นั้นจะเก็บข้อมูลแตกต่างกันไป ขึ้นกับว่า node นั้น ๆ เป็น internal node หรือ leaf node

- **leaf node** เก็บ substring และเก็บค่า weight ซึ่งมีค่าเท่ากับความยาวของ substring ที่ node นั้นเก็บไว้
- **internal node** เก็บค่า weight ซึ่ง weight มีค่าเท่ากับผลรวม weight ของ leaf node ทั้งหมดใน left subtree ของ node นั้น ๆ ซึ่งเท่ากับความยาวรวมของ substring ใน left subtree

rope จะทำการเก็บข้อมูลคล้าย ๆ กับ binary search tree แต่แทนที่จะเก็บ node ที่มีค่าน้อยกว่าไว้ที่ลูกทางซ้ายและเก็บ node ที่มีค่ามากกว่าไว้ที่ลูกทางขวา rope จะเก็บข้อมูลของ string ที่ตำแหน่งน้อยไว้ทางซ้ายและเก็บข้อมูลของ string ที่ตำแหน่งมากกว่าไว้ทางขวาแทน

## Operations

1. **`toString()`** สร้าง string จาก rope `O(n)`

   การสร้าง string จาก rope สามารถทำได้โดยการ traverse ใน tree แล้วนำ leaf node มาต่อกันจากซ้ายสุดไปขวาสุด

```
void toString(node* ptr, string& str) {
  if (!ptr) return;
  if (ptr->isLeaf()) {
    str.append(ptr->str); return;
  }
  toString(ptr->left, str);
  toString(ptr->right, str);
}
```

2. **`length()`** หาความยาวของ rope `O(log n)`

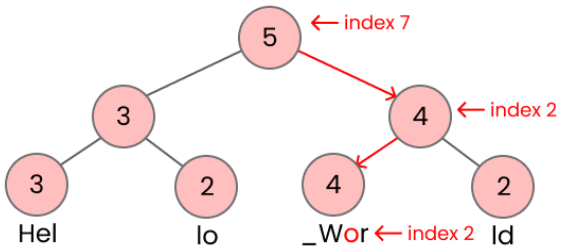
   สามารถหาได้จากการนำความยาวรวมของ substring ใน left subtree ซึ่งก็คือ weight ของ node ลูกทางซ้าย มาบวกกับความยาวรวมของ substring ใน right subtree ซึ่งสามารถหาได้จากการ recursive

```
size_t length(node* ptr) {
  if (!ptr) return 0;
  return ptr->weight + length(ptr->right);
}
```

3. **`index(i)`** เรียกดูค่าของตัวอักษรในตำแหน่งที่ `i` ของ rope `O(log n)`

   จากการที่แต่ละ node เก็บค่าความยาวของ substring ใน left subtree ไว้นั้น ทำให้สามารถรู้ได้ว่าตัวอักษรลำดับที่ `i` จะอยู่ใน left หรือ right subtree ดังนั้นจึงสามารถใช้การ recursive ในการหาตัวอักษรที่ต้องการได้

   หาก `i < weight` ของ node นั้น ๆ แสดงว่าตัวอักษรที่ต้องการ เป็นตัวอักษรตัวที่ `i` ของ left subtree แต่หาก `i >= weight` แสดงว่าตัวอักษรที่ต้องการ เป็นตัวอักษรตัวที่ `i - weight` ของ right subtree และหาก node ปัจจุบันเป็น leaf node แล้ว เราก็สามารถเรียกค่าของตัวอักษรตัวที่ `i` ใน substring ได้เลย




ตัวอย่างการเรียก `index(7)`

```
char findIndex(node* ptr, size_t index) {
   if (ptr->isLeaf()) return ptr->str[index];
   if (index < ptr->weight)
     return findIndex(ptr->left, index);
   else
     return findIndex(ptr->right, index - ptr->weight);
 }
```

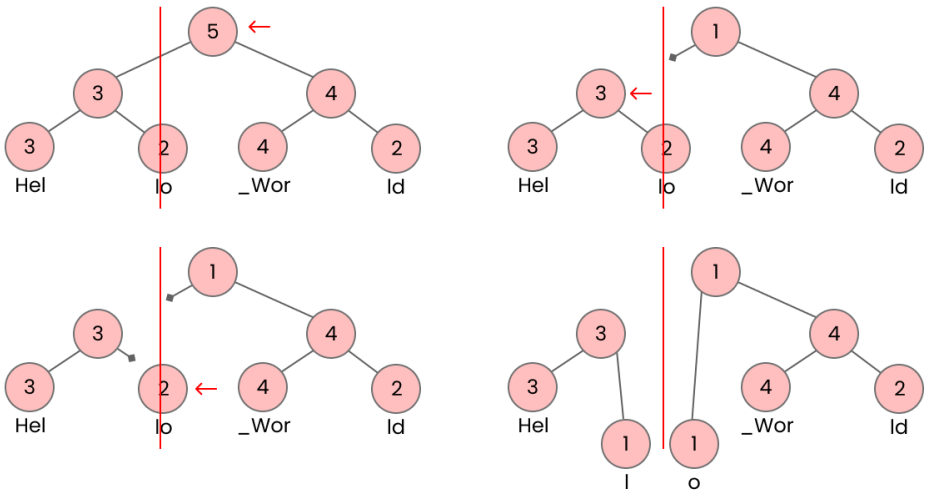
4. **`concat(r1, r2)`** นำ rope `r1` และ `r2` มาต่อกัน `O(log n)`

   การนำ rope มาต่อกันนั้น สามารถทำได้ง่าย ๆ โดยการสร้าง node ใหม่ขึ้นมา แล้วให้ **`r1`** เป็น left subtree และให้ **`r2`** เป็น right subtree จากนั้นจึงคำนวณ weight ของ node ที่สร้างขึ้นมาใหม่

5. **`split(i)`** แบ่ง rope ออกเป็นสองส่วนที่ตำแหน่ง **`i`** `O(log n)`

   การแบ่ง rope เป็นสองส่วนสามารถทำได้โดยการ recursive โดยทำการดูว่าตำแหน่งที่ต้องถูกแบ่งอยู่ใน left หรือ right subtree หาก **`i < weight`** แสดงว่าตำแหน่งที่ต้องแบ่งต่ออยู่ใน left subtree ซึ่งหมายความว่า node ที่กำลังพิจารณาอยู่นั้นจะอยู่ใน rope ผลลัพธ์ส่วนขวา ในอีกด้านหนึ่ง หาก **`i >= weight`** แสดงว่าตำแหน่งที่ต้องแบ่งต่ออยู่ใน right subtree หมายความว่า node ที่กำลังพิจารณาจะอยู่ใน rope ผลลัพธ์ส่วนซ้าย จากนั้นจึงทำการ recursive เพื่อแบ่ง subtree ต่อไป โดยต้องไม่ลืมปรับค่า weight ของ node นั้น ๆ ด้วย

   เมื่อทำการ recursive มาจนถึง leaf node แล้ว จึงค่อยทำการแบ่ง leaf node นั้นเป็น leaf node สองอันใหม่ โดยการแบ่ง string ใน leaf node นั้นเป็นสองส่วน แล้วจึงแบ่ง leaf node สองอันนั้นไปอยู่ใน rope ผลลัพธ์ส่วนซ้ายและขวา




ตัวอย่างการเรียก `split(4)`

```
void split(node* ptr, node*& left, node*& right, size_t index) {
  if (!ptr) return;
  if (ptr->isLeaf()) {
    left = new node(ptr->str.substr(0, index));
    right = new node(ptr->str.substr(index));
  } else if (index < ptr->weight) {
    split(ptr->left, left, ptr->left, index);
    ptr->weight -= index;
    right = ptr;
  } else {
    split(ptr->right, ptr->right, right, index - ptr->weight);
    left = ptr;
  }
}
```

6. **insert(i, r)** แทรก rope r เข้าไปก่อนหน้าตัวอักษรลำดับที่ **i** O(log n)

การแทรก rope สามารถทำได้โดยการแบ่ง rope ออกเป็นส่วนซ้ายและขวาที่ตำแหน่ง **i** จากนั้นจึงนำส่วนซ้ายมาเชื่อมเข้ากับ rope ที่เราต้องการแทรก แล้วจึงนำไปเชื่อมกับส่วนขวา

```
void insert(size_t index, rope r) {
  node *left, *right;
  split(mRoot, left, right, index);
  mRoot = concat(left, r.mRoot);
  mRoot = concat(mRoot, right);
}
```

7. **delete(i, j)** ลบตัวอักษรตั้งแต่ index **i** จนถึง ก่อนหน้า index **j** ออก O(log n)

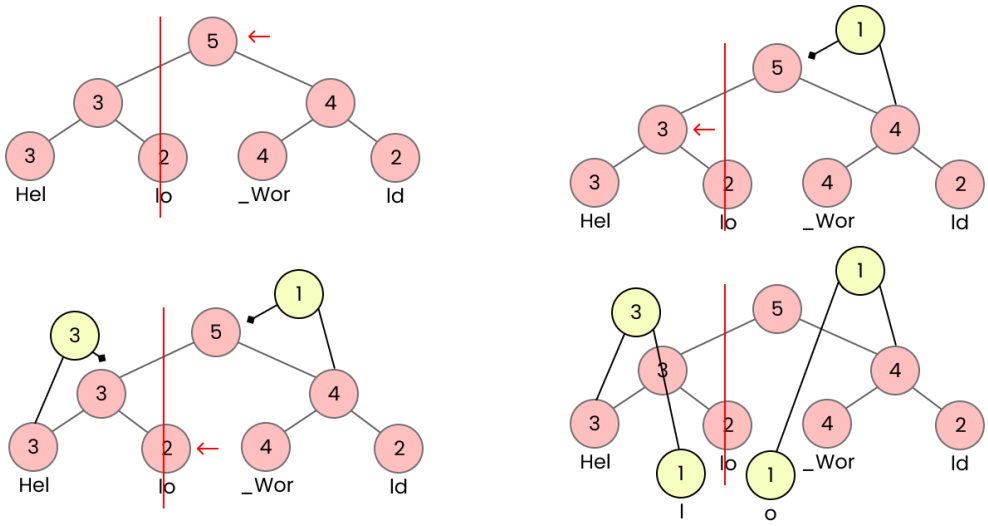
การลบ substring สามารถทำได้โดยการแบ่ง rope เป็นสามส่วนที่ตำแหน่ง **i** และตำแหน่ง **j** แล้วจึงนำส่วนซ้ายและส่วนขวากลับมาเชื่อมกันใหม่โดยทิ้งส่วนกลางไป

```
void erase(size_t st, size_t ed) {
  node *left, *mid, *right;
  split(mRoot, left, right, ed);
  split(left, left, mid, st);
  mRoot = concat(left, right);
}
```

## การ undo และ redo

สังเกตว่าการทำ operation ตามที่อธิบายในข้างต้นนั้น การ split จะทำให้ rope เดิมถูกทำลายไป เนื่องจากมีการเปลี่ยนแปลง left child, right child และ weight ของ node ต่าง ๆ หากทำการเปลี่ยนวิธี split ให้ไปเป็นการสร้าง node ใหม่ แทนที่จะแก้ไขข้อมูลของ node เดิมเลย จะทำให้ข้อมูลของ rope เดิมยังคงอยู่ หากเราเก็บ root node ของ rope ใน version ต่าง ๆ ไว้ จะทำให้สามารถย้อนไปดูข้อมูลของ rope ในแต่ละ version ได้

การที่เราไม่ทำลาย rope เดิมนั้น นอกจากจะทำให้เราสามารถเก็บ version ต่าง ๆ ได้แล้ว ยังสามารถทำให้ rope หลาย ๆ อันสามารถใช้ node ร่วมกันได้ ซึ่งสามารถประหยัด memory และประหยัดเวลาในการ copy ได้อีกด้วย



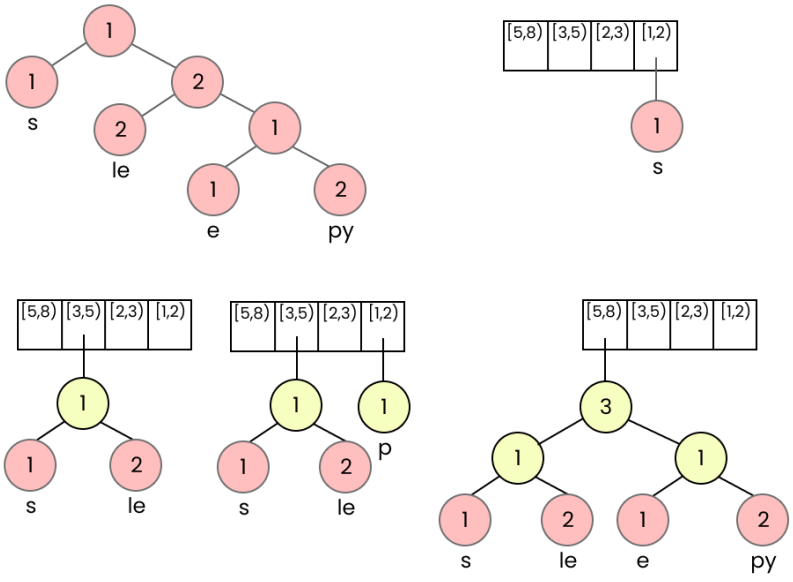

ตัวอย่างการเรียก `split(4)` โดย node สีเหลืองเป็น node ที่ถูกสร้างขึ้นใหม่

```
void split(node* ptr, node*& left, node*& right, size_t index) {
  if (ptr->isLeaf()) {
    left = new node(ptr->str.substr(0, index));
    right = new node(ptr->str.substr(index));
  } else if (index < ptr->weight) {
    right = new node(NULL, ptr->right, ptr->weight - index);
    split(ptr->left, left, right->left, index);
  } else {
    left = new node(ptr->left, NULL, ptr->weight);
    split(ptr->right, left->right, right, index - ptr->weight);
  }
}
```

## Rebalance

เนื่องจากเวลาที่ใช้ในการทำงานของ operation ต่าง ๆ นั้นขึ้นอยู่กับความสูงของ rope เมื่อความสูงของ rope มากก็ควรทำการ rebalance เพื่อที่จะลดเวลาที่ใช้ในการทำ operation ต่าง ๆ โดยการ rebalance นั้นสามารถทำในรูปแบบเดียวกับ self-balancing binary search tree ปกติ เช่น AVL tree ก็ได้ แต่วิธีที่ถูกเสนอขึ้นมาในการ rebalance rope นั้น คือการใช้ Fibonacci sequence โดย rope ที่มีความลึก n นั้นจะ balance เมื่อ rope นั้นมีความยาวไม่ต่ำกว่า $F_{n+2}$ (rope ที่ balance นั้น ไม่จำเป็นต้องมีลูกซ้ายและลูกขวาที่ balance) การ rebalance rope แบบนี้นั้นจะสร้าง rope version ใหม่โดยที่ไม่ทำลาย rope เดิม ทำให้ยังสามารถย้อนไปดูข้อมูลของ rope ใน version ก่อน ๆ ได้

การ rebalance rope นั้นจะทำโดยการค่อย ๆ สร้าง rope ขึ้นใหม่จาก leaf เดิมทั้งหมดจากซ้ายไปขวา ในการสร้าง rope ใหม่นั้น เราจะทำการ maintain ลำดับของ tree หลาย ๆ อัน โดยสำหรับทุก ๆ leaf ที่เราพิจารณานั้น หาก weight ของ leaf อยู่ในช่วง $[F_n, F_{n+1})$ แสดงว่า leaf นั้นควรจะถูกใส่ไปที่ลำดับที่ n ของ sequence แต่หากมี tree ลำดับน้อยกว่านั้นอยู่ ก็ต้องทำการ concatenate tree ที่ลำดับน้อยกว่าทั้งหมดก่อนแล้วจึงนำมา concatenate กับ leaf ใหม่ และหากมี tree ที่ลำดับที่ n อยู่ก็ต้องนำผลลัพธ์ที่ได้ไป concatenate กับ tree ในลำดับที่ n แล้วจึงนำผลลัพธ์ไปใส่ในลำดับที่ n+1 แทน หากยังมี tree ในลำดับที่ n+1 อีก ก็ต้อง concatenate ไปเรื่อย ๆ จนกว่าจะมีลำดับที่ว่างอยู่ เมื่อทำเสร็จครบทุก leaf แล้ว จึงนำ tree ทั้งหมดใน sequence มา concatenate กัน ก็จะได้เป็น rope ใหม่ที่ balance




ตัวอย่างการ rebalance rope (rope เริ่มต้นนั้น balance อยู่แล้ว แต่ยกมาเป็นตัวอย่างเพื่อให้เข้าใจง่าย)

```
typedef priority_queue<pair<int, node*>, vector<pair<int, node*>>,
                       greater<pair<int, node*>>>
    rebalancepq;

void rebalance() {
  if (length() >= fibo.get(mRoot->depth + 2)) return;
  rebalancepq pq;
  rebalance(pq, mRoot);
  mRoot = NULL;
  while (!pq.empty()) {
    mRoot = concat(pq.top().second, mRoot);
    pq.pop();
  }
  if (!mRoot) mRoot = new node();
}

void rebalance(rebalancepq& pq, node* ptr) {
  if (!ptr) return;
  if (ptr->isLeaf()) {
    if (ptr->weight == 0) return;
    int index = fibo.getIndex(ptr->weight);
    node* tmp = NULL;
    while (!pq.empty() && pq.top().first < index) {
      tmp = concat(pq.top().second, tmp);
      pq.pop();
    }
    ptr = concat(tmp, ptr);
    while (!pq.empty() && pq.top().first == index) {
      ptr = concat(pq.top().second, ptr);
      index++;
      pq.pop();
    }
    pq.push(make_pair(index, ptr));
    return;
  }
  rebalance(pq, ptr->left);
  rebalance(pq, ptr->right);
}
```

Rope C++ implementation:

https://github.com/PKhing/INTRO_DATA_STRUCT/blob/main/homework/rope.h

## บรรณานุกรม


1. Ropes: an Alternative to Strings https://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.14.9450&rep=rep1&type=pdf
2. Rope: the Data Structure used by text editors to handle large strings, https://iq.opengenus.org/rope-data-structure/
3. Rope (data structure) https://en.wikipedia.org/wiki/Rope_(data_structure)
4. Ropes — Fast Strings https://kukuruku.co/post/ropes-fast-strings/
5. Ropes Data Structure (Fast String Concatenation) https://www.geeksforgeeks.org/ropes-data-structure-fast-string-concatenation/
6. Rope Data Structure https://medium.com/underrated-data-structures-and-algorithms/rope-data-structure-e623d7862137
7. Ruby Conference 2007 Ropes: An Alternative to Ruby's Strings by Eric Ivancich https://www.youtube.com/watch?v=5Xt6qN269Uo