Problem Set 02

Mostafa Touny

Contents

Exercises

Ex. 2-1

done

Ex. 2-2

For our own convenience of avoiding tedious computations, we multiply $A(x) = -10 + x$ with $B(x) = 3 - 6x$.

Double-degree form

$A(x) = -10 + x + 0x^2 + 0x^3$

$B(x) = 3 - 6x + 0x^2 + 0x^3$

Computing A(x) on sample

```
Recursive-FFT(-10, 1, 0, 0)
  n = 4
  w_4 = e^{2 pi i / 4}
  w = w_4^0 = 1
  a[even] = (-10, 0)
  a[odd] = (1, 0)
  y[even] = Recursive-FFT(-10, 0) = (-10, -10)
  y[odd] = Recursive-FFT(1, 0) = (1, 1)
  for k=0 to 1
    k=0
      y_0 = (-10) + (1)(1) = -9
      y_2 = (-10) - (1)(1) = -11
      w = w_4^1
    k=1
      y_1 = (-10) + (e^{1 2 pi i / 4})(1) = -10+i
      y_3 = (-10) - (e^{1 2 pi i / 4})(1) = -10-i
      w = w_4^2
  return (-9, -10+i, -11, -10-i)
```

```
Recursive-FFT(-10, 0)
  n = 2
  w_2 = e^{2 pi i / 2}
  w = w_2^0 = 1
  a[even] = (-10)
  a[odd] = (0)
  y[even] = Recursive-FFT(-10) = (-10)  // base case
  y[odd] = Recursive-FFT(0) = (0)
  for k=0 to 0
    k=0
      y_0 = (-10) + w (0) = -10
      y_1 = (-10) - w (0) = -10
      w = w_2^1
  return (-10, -10)
```

```
Recursive-FFT(1, 0)
  n = 2
  w_2 = e^{2 pi i / 2}
```

```
  w = w_2^0 = 1
  a[even] = (1)
  a[odd] = (0)
  y[even] = Recursive-FFT(1) = (1) // base case
  y[odd] = Recursive-FFT(0) = (0)
  for k=0 to 0
    k=0
      y_0 = 1 + w(0) = 1
      y_1 = 1 - w(0) = 1
      w = w_2^1
  return (1, 1)
```

Computing B(x) on sample

Similarly, we get $y = (-3, 3 - 6i, 9, 3 + 6i)$

Computing C(x) on sample, By multiplying corresponding sample points of A and B

$y = ((-9)(-3), (-10 + i)(3 - 6i), (-11)(9), (-10 - i)(3 + 6i)) = (27, -24 + 63i, -99, -24 - 63i)$

Interpolating C(x) coefficients

```
Recursive-IFFT(27, -24+63i, -99, -24-63i)
  n = 4
  w_4^-1 = e^{-1 i 2 pi / 4}
  w = w_4^0 = 1
  y[even] = (27, -99)
  y[odd] = (-24+63i, -24-63i)
  a[even] = Recursive-IFFT(27, -99) = (-72, 126)
  a[odd] = Recursive-IFFT(-24+63i, -24-63i) = (-48, 126i)
  for k=0 to 1
    k=0
      y_0 = (-72) + (1)(-48) = -120
      y_2 = (-72) - (1)(-48) = -24
      w = w_4^-1

    k=1
      y_1 = (126) + (e^{-1 i 2 pi / 4})(126i) = 252
      y_3 = (126) - (e^{-1 i 2 pi / 4})(126i) = 0
      w = w_4^-2
  return (-120, 252, -24, 0)
```

```
Recursive-IFFT(27, -99)
  n = 2
  w_2^-1 = e^{-1 i 2 pi / 2}
  w = w_2^0 = 1
  y[even] = (27)
  y[odd] (-99)
  a[even] = Recursive-IFFT(27) = (27)  // base case
  a[odd] = Recursive-IFFT(-99) = (-99)
  for k=0 to 0
    k=0
      y_0 = 27 + (1)(-99) = -72
      y_1 = 27 - (1)(-99) = 126
      w = w_2^-1
```

```
  return (-72, 126)

Recursive-IFFT(-24+63i, -24-63i)
  n = 2
  w_2^-1 = e^{-1 i 2 pi /2}
  w = w_2^0 = 1
  y[even] = (-24+63i)
  y[odd] = (-24-63i)
  a[even] = Recursive-IFFT(-24+63i) = (-24+63i)
  a[odd] = Recursive-IFFT(-24-63i) = (-24-63i)
  for k=0 to 0
    k=0
      y_0 = (-24+63i) + (1)(-24-63i) = -48
      y_1 = (-24+63i) - (1)(-24-63i) = 126i
  return (-48, 126i)
```

Hence, Final answer is (-120, 252, -24, 0)/4 = (-30, 63, -6, 0), and resulting polynomial is $C(x) = -30 + 63x - 6x^2$.

Ex. 2-3

Modifying `Recursive-FFT`, by switching a and y, replacing w_n by w_n^{-1}. Finally, result vector is divided by n.

```
Recursive-IFFT(y)
  n = y.length
  if n == 1
    return y
  w_n^-1 = e^{-1 2 pi i / n}
  w = 1
  y[even] = (y_0, y_2, ..., y_n-2})
  y[odd] = (y_1, y_3, ..., y_n-1)
  a[even] = Recursive-IFFT(y[even])
  a[odd] = Recursive-IFFT(y[odd])
  for k=0 to n/2 - 1
    a_k = a[even]_k + w a[odd]_k
    a_k+(n/2) = a[even]_k - w a[odd]_k
    w = w w_n^-1
  return a

a = a/n
```

Ex. 2-4

done

Ex. 2-5

Create operation accounts for number of pointers filled. Insert operations are modified to allow up to $2t - 1$ keys in case of internal node, and up to $(2t - 1) + (2t) = 4t - 1$ keys in case of leaf node. That, by basically modifying the if condition. Also, insertion in place of pointers happens by checking whether a leaf have $2t - 1$ keys.

Note we haven't rigorously proven our modification is correct; We rely on our intuition to write main parts of new the operations.

```
B-TREE-CREATE(T)
  x = ALLOCATE-NODE()
  x.leaf = TRUE
  x.n = 0
  x.n' = 0  // number of pointers to children filled
  DISK-WRITE(x)
  T.root = x

B-TREE-INSERT(T,k)
  r = T.root
  if (r.n == 2t-1 and not r.leaf) or (r.n == 4t-1 and r.leaf)  // different cases for internal and le
    s = ALLOCATE-NODE()
    T.root = s
    s.leaf = FALSE
    s.n = 0
    s.c1 = r
    B-TREE-SPLIT-CHILD(s,1)
    B-TREE-INSERT-NONFULL(s,k)
  else
    B-TREE-INSERT-NONFULL(r,k)

B-TREE-INSERT-NONFULL(x,k)
  if x.leaf
    i = x.n + x.n'  // sum of both keys and pointers

    while i >= x.n + 1 and k < x.c_(i-x.n)
      x.c_(i-x.n+1) = x.c_(i-x.n)
      i = i-1
    while i >= 1 and k < x.key_i
      x.key_i+1 = x.key_i
      i = i-1

    if i >= x.n + 1
      x.c_i+1 = k
      x.n' = x.n' + 1
    else
      x.key_i+1 = k
      x.n = x.n + 1

    DISK-WRITE(x)

  else
    i = x.n

    while i>=1 and k<x.key_i
      i = i-1
    i = i+1

    DISK-READ(x,c_i)
```

```
    if (x.c_i).n == 2t-1  // note this is an internal node
      B-TREE-SPLIT-CHILD(x,i)
      if k > x.key_i
        i = i+1

    B-TREE-INSERT-NONFULL(x.c_i, k)
```

Ex. 3-2

We implement the prescription described in p.500.

Note we haven't rigorously proven our modification is correct; We rely on our intuition to write main parts of new the operations.

```
B-TREE-DELETE(x, k)
  // check if k is in node x
  i = x.n
  while i>=1 and x.key_i != k
    i = i-1

  // k is found
  if i>=1

    // x is a leaf node
    if x.leaf
      key_i = NULL
      x.n = x.n - 1
      x.shiftKeysAndPointers()  // for brevity we ignore implementing this subroutine

    // x is an internal node
    else
      y = x.c_i  // child preceeding k
      z = x.c_i+1  // child following k

      // number of keys in child preceeding k is at least t
      if y.n >= t
        k' = y.lastKey()  // implementation is ignored
        B-TREE-DELETE(y, k')
        key_i = k = k'

      // number of keys in child following k is at least t
      else if z.n >= t
        // symmetrically replace k by k'

      // number of keys in both child following and preceeding k, is less than t
      else
        mergeInto([y, k, z])  // merge k and z into y. implementation is ignored
        x.c_i+1 = NULL
        B-TREE-DELETE(k)


  // k isn't found in x
  else
```

```
  // find k in children
  i = x.n
  while i >= 1 and k < x.key_i
    i = i-1
  y = x.c_(i+1) // subtree y containing k

  // guarantee we descend to a node containing at least t keys
  if y.n == t-1

    if i+2 <= x.n
      z = x.c_(i+2)  // immediate forward sibling of y
    if i >= 1
      r = x.c_(i)  // immediate preceeding sibling of y

    // some sibling contains at least t keys
    if z and z.n >= t
      B-TREE-INSERT(y, key_i)
      B-TREE-DELETE(key_i)
      B-TREE-INSERT(x, z.firstKey())
      B-TREE-DELETE(z, z.firstKey())
      // don't get why a pointer from sibling should be moved to y
    else if r and r.n >= t
      // symmetrically

    // both immediate siblings have t-1 keys
    else
      mergeInto([r, key_i, y])
      x.c_(i+1) = NULL
      B-TREE-DELETE(x, key_i)

  // remove k from child y
  B-TREE-DELETE(y, k)
```

Problems

Problem 2-1

a

```
Match(S, P)
  n = S.length
  m = P.length

  M = []

  for i = 0..n-m+1
    flag = true
    for j = 0..m
      if P[j] != S[i+j] flag = false
    if flag M.append(i)

  return M
```

b

Source string S gets encoded as $S(x) = s_0x^0 + s_1x^1 + \cdots + s_{n-1}x^{n-1}$, and pattern string P as $P(x) = p_0x^{m-1} + p_1x^{m-2} + \cdots + p_{m-1}x^0$, where s_i and p_i are, 1 or -1, if characters $S[i]$ and $P[i]$, are a or b, respectively. If $P[i] = *$, Then $p_i = 0$.

Observe $s_jp_k = 1$ if $S[j] = P[k]$, $s_jp_k = -1$ if $S[j] \neq P[k]$, and $s_jp_k = 0$ if $P[k] = *$. Observe for resulting polynomial $(s \cdot p)(x) = r_0x^0 + r_1x^1 + \cdots + r_{m+n-2}x^{m+n-2}$, Coefficient $r_i = \sum_{j+k=i} s_jp_k$, Exactly matches the sum of multiplying s_{i-m+1}, s_{i-m+2}, …, s_{i-1}, s_i with p_0, p_1, …, p_{m-1}, respectively, for $i = m-1, m, \ldots, n-1$. If and only if, All corresponding alphabetic characters are equal, Then each contributes to the sum by $+1$. Asterik $*$ always contributes nothing to the sum. Therefore, if k is the number of alphabetic character in P (non asterik characters), Then $r_i = k$ if and only if P matches substring $S[i-m+1..i]$.

Now we can set output M to be the ordered list of i such that $r_i = k$, Then subtract each entry by $m-1$, so that i matches the position of the first character of the substring.

Note coefficients r_0, r_1, …, r_{m-2} are irrelevant to our consideration, Since they do not consider a matching with the whole characters of pattern string P.

For the example, $S = ababbab$ and $P = ab*$,

$$\begin{aligned}
S(x) &= (1)x^0 + (-1)x^1 + (1)x^2 + (-1)x^3 + (-1)x^4 + (1)x^5 + (-1)x^6 \\
P(x) &= (1)x^2 + (-1)x^1 + (0)x^0 \\
(S \cdot P)(x) &= (-1)x^1 + (x)x^2 + (-2)x^3 + (2)x^4 + (1)x^7 + (-1)x^8 \\
M &= [2, 4] \\
M &= [2-(m-1), 4-(m-1)] = [2-(3-1), 4-(3-1)] \\
M &= [0, 2]
\end{aligned}$$

c

$\mathcal{O}(n \lg n)$, Since each operation of my algorithm requires at most a linear scan of complexity $\mathcal{O}(n)$

d

Exactly as b, but characters are encoded as

	A	C	G	T	*
S	1	-1	i	-i	
P	1	-1	-i	i	0

Note if characters are matching, Then as before, each contributes to the sum by $+1$. In case of non-matching, A number less than $+1$ or an imaginary number is contributed.

Problem 2-2

a

Merge roots of T_1 and T_2, placing k in between them. If new root's keys are greater than $2t-1$, Apply the standard operation of split and push median up.

Note roots of T_1 and T_2, each has at most $2t-1$ keys. When merging the new root is at most $(2t-1) + (2t-1) + 1 = 4t-1$. If we spllitted, We get a new root of key size exactly 1 and two childs, Each is of size at most $2t-1$.

b

Modify T_1 to decrease its height by one, Then apply the same procedure of a. That, by merging all of T_1's children into its root. In other words, $Merge(x.c_0, key_0, x.c_1, key_1, \ldots, x.c_{n-1}, key_{n-1}, x.c_n)$. Note new root of T_1 has at most $(2t-1)+(2t)(2t-1) = (2t+1)(2t-1) = 4t^2-1$ keys.

As in a, Merge T_1, k, and T_2. The new root has at most $(4t^2-1)+(2t-1)+1 = (2t+2)(2t-1)+1 = 4t^2+2t-1$ keys. Hence we are going to split around $\mathcal{O}(\lg t^2) = \mathcal{O}(\lg t)$ times. Note that shall result in many one-key nodes. So, at most $\mathcal{O}(\lg t)$ merge of one-key nodes, to finally fix the tree. But since t is assumed to be a constant, the total complexity is $\mathcal{O}(1)$.

c

The tree's height is increased only when the root has a full capacity of keys, and a new root is allocated.

```
B-TREE-INSERT(T,k)
r = T.root
if r.n == 2t-1
  s = ALLOCATE-NODE()
  T.root = s
  s.leaf = FALSE
  s.n = 0
  s.c1 = r
  s.height = r.height+1  // new root is of height +1 than the previous one
  B-TREE-SPLIT-CHILD(s,1)
  B-TREE-INSERT-NONFULL(s,k)
else
  B-TREE-INSERT-NONFULL(r,k)
```

Tree shrinks only when a merge happens with the root alongside its children. If we assumed the new root is updated to point to one of the children, Then no height variable needs to be updated.

d

Without the loss of generality assume $h_1 \geq h_2$. Then the procedure of b is applied $h_1 - h_2$ times so T_1 and T_2 have the same weight. Then the procedure of a is applied to combine them. Each operation costs a constant time, Hence a complexity upperbounded by $\mathcal{O}((h_1 - h_2) + 1)$.