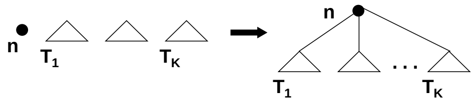
תרגול מס׳ 4
עצים

פרק 12 במבוא לאלגוריתמים / קורמן

A hierarchical combinatorial structure

הגדרה רקורסיבית:

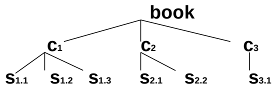
1. צומת בודד. זהו גם שורש העץ.
2. אם n הוא צומת ו $T_1....T_K$ הינם עצים, ניתן לבנות עץ חדש שבו n השורש ו $T_1....T_K$ הינם "תתי עצים".



מושגים:

Example : description of a book

book
c1
s1.1
s1.2
s1.3
c2
s2.1
s2.2
c3
s3.1



מושגים:

book - **הורה/Parent** (אבא) של c1, c2, c3

c_1, c_2 - **ילדים/children** של book

$s_{2.1}$ - **צאצא/Descendant** (לא ישיר) של book

$book, c_1, s_{1.2}$ - **מסלול/Path** (אם כ"א הורה של הקודם)

אורך המסלול = מס׳ הקשתות
= מס׳ הצמתים (פחות אחד)

צומת ללא ילדים = **עלה/Leaf**

book - **אב קדמון/Ancestor** של $s_{3.1}$

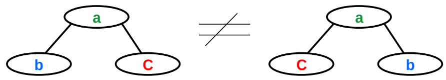
9

<u>גובה העץ</u> - אורך המסלול הארוך ביותר מהשורש לעלה (height)

<u>עומק צומת</u> - אורך המסלול מהצומת לשורש (depth)

<u>Ordered tree</u>

יש משמעות לסדר הילדים. מסדרים משמאל לימין.



אם הסדר לא חשוב - <u>עץ לא מסודר (unordered tree)</u>

עצים בינאריים

- עץ ריק או לכל צומת יש תת קבוצה של {ילד ימני, ילד שמאלי}

דוגמא:

עץ מלא: לכל צומת פנימית יש תמיד שני ילדים

The dictionary problem

- Maintain (distinct) items with keys from a totally ordered universe subject to the following operations

The ADT

- Insert(x,D)
- Delete(x,D)
- Find(x,D):
  Returns a pointer to x if x ∈ D, and a pointer to the successor or predecessor of x if x is not in D

The ADT

- successor(x,D)
- predecessor(x,D)
- Min(D)
- Max(D)

The ADT

- catenate(D_1,D_2) : Assume all items in D_1 are smaller than all items in D_2
- split(x,D) : Separate to D_1, D_2
  - D_1 with all items greater than x and
  - D_2 smaller than x

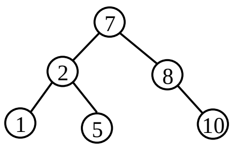
Reminder from "mavo"

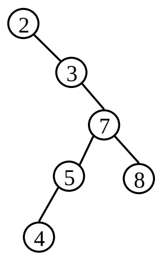
- We have seen solutions using unordered lists and ordered lists.
- Worst case running time O(n)
- We also defined Binary Search Trees (BST)

Binary search trees

- A representation of a set with keys from a totally ordered universe
- We put each element in a node of a binary tree subject to:

BST

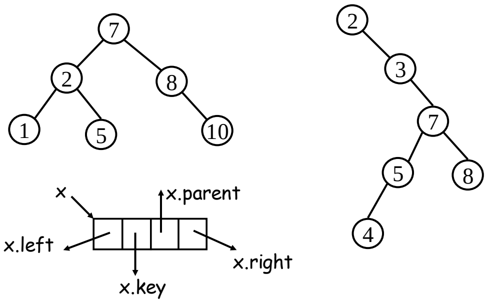




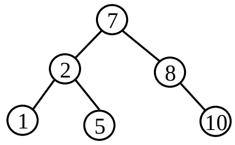
If y is in the left subtree of x then y.key < x.key

If y is in the right subtree of x then y.key > x.key

BST

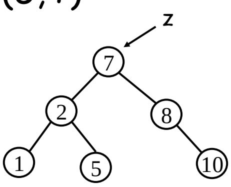


Find(x,T)



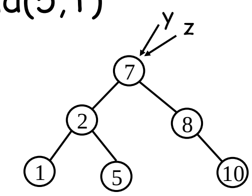
y ← null
z ← T.root
While z ≠ null
    do y ← z
        if x = z.key return z
        if x < z.key then z ← z.left
                                else z ← z.right
return y

Find(5,T)



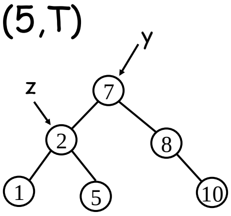
y ← null
z ← T.root
While z ≠ null
    do y ← z
        if x = z.key return z
        if x < z.key then z ← z.left
                                else z ← z.right
return y

Find(5,T)



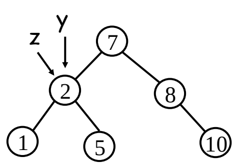
y ← null
z ← T.root
While z ≠ null
    do y ← z
        if x = z.key return z
        if x < z.key then z ← z.left
                                else z ← z.right
return y

Find(5,T)



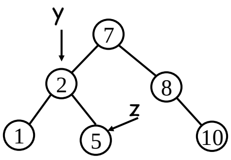
y ← null
z ← T.root
While z ≠ null
    do y ← z
        if x = z.key return z
        if x < z.key then z ← z.left
                                else z ← z.right
return y

Find(5,T)



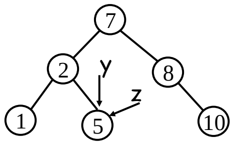
y ← null
z ← T.root
While z ≠ null
    do y ← z
        if x = z.key return z
        if x < z.key then z ← z.left
                                else z ← z.right
return y

Find(5,T)



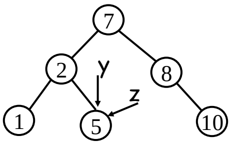

```
Y ← null
z ← T.root
While z ≠ null
     do y ← z
          if  x = z.key return z
          if  x < z.key then z ← z.left
                              else z ← z.right
return y
```


Find(5,T)



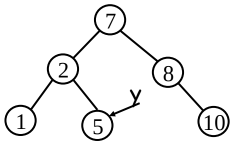

```
Y ← null
z ← T.root
While z ≠ null
     do y ← z
          if  x = z.key return z
          if  x < z.key then z ← z.left
                              else z ← z.right
return y
```


Find(6,T)




```
Y ← null
z ← T.root
While z ≠ null
     do y ← z
          if  x = z.key return z
          if  x < z.key then z ← z.left
                              else z ← z.right
return y
```

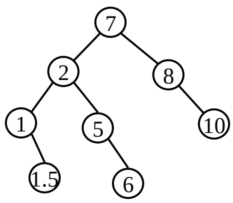

Find(6,T)



z=null

```
Y ← null
z ← T.root
While z ≠ null
     do y ← z
          if  x = z.key return z
          if  x < z.key then z ← z.left
                              else z ← z.right
return y
```


Min(T)

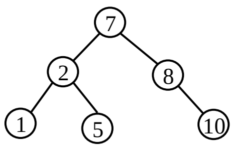


Min(T.root)

min(z):

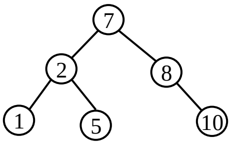
```
While (z.left ≠ null)
     do z ← z.left
return (z)
```


Insert(x,T)



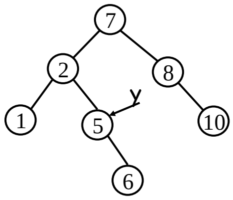

```
n ← new node
n.key←x
n.left ← n.right ← null
y ← find(x,T)
n.parent ← y
if  x < y.key then y.left ← n
                 else y.right ← n
```

Insert(6,T)



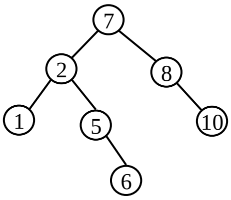

```
n ← new node
n.key←x
n.left ← n.right ← null
y ← find(x,T)
n.parent ← y
if  x < y.key then y.left ← n
                else y.right ← n
```


Insert(6,T)

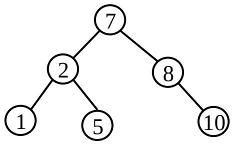



```
n ← new node
n.key←x
n.left ← n.right ← null
y ← find(x,T)
n.parent ← y
if  x < y.key then y.left ← n
                else y.right ← n
```

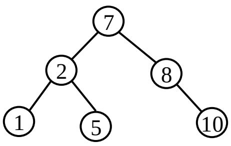

Delete(6,T)



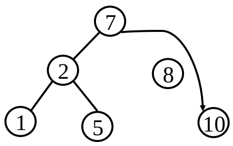
Delete(6,T)



Delete(8,T)

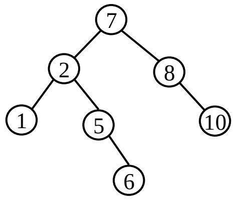


Delete(8,T)



Delete(2,T)

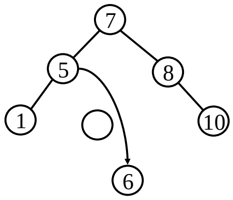
Switch 5 and 2 and
delete the node
containing 5



33

Delete(2,T)

Switch 5 and 2 and
delete the node
containing 5

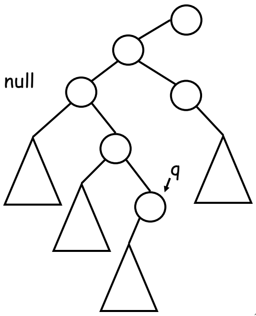


34

delete(x,T)

q ← find(x,T)

If q.left = null or q.right = null
then z ← q
else z ← min(q.right)
q.key ← z.key

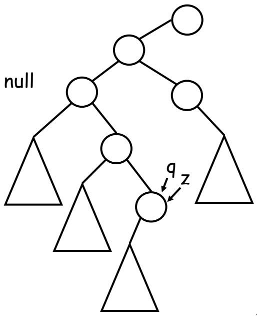


35

delete(x,T)

q ← find(x,T)

If q.left = null or q.right = null
then z ← q
else z ← min(q.right)
q.key ← z.key

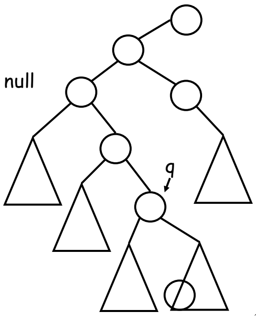


36

delete(x,T)

q ← find(x,T)

If q.left = null or q.right = null
then z ← q
else z ← min(q.right)
q.key ← z.key

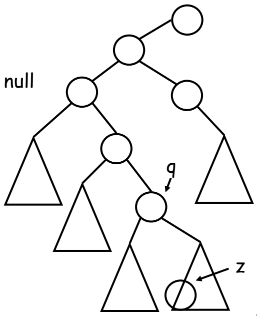


37

delete(x,T)

q ← find(x,T)

If q.left = null or q.right = null
then z ← q
else z ← min(q.right)
q.key ← z.key

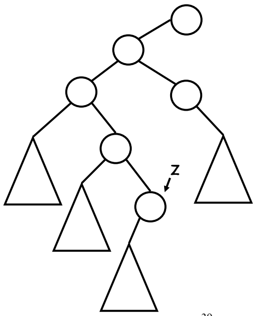


38

delete(x,T)

q ← find(x,T)

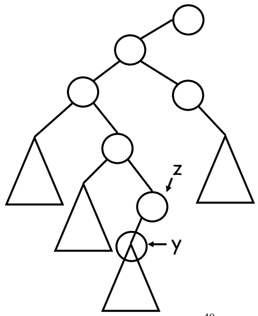
If q.left = null or q.right = null
then z ← q
else z ← min(q.right)
    q.key ← z.key
If z.left ≠ null then y ← z.left
    else y ← z.right



delete(x,T)

q ← find(x,T)

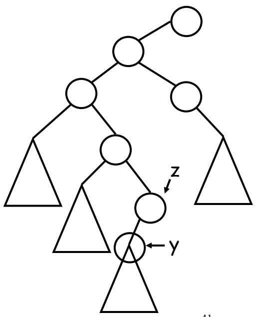
If q.left = null or q.right = null
then z ← q
else z ← min(q.right)
    q.key ← z.key
If z.left ≠ null then y ← z.left
    else y ← z.right



delete(x,T)

q ← find(x,T)

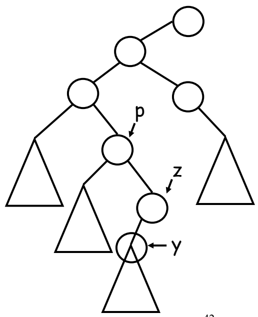
If q.left = null or q.right = null
then z ← q
else z ← min(q.right)
    q.key ← z.key
If z.left ≠ null then y ← z.left
    else y ← z.right
If y ≠ null then
    y.parent ← z.parent



delete(x,T)

q ← find(x,T)

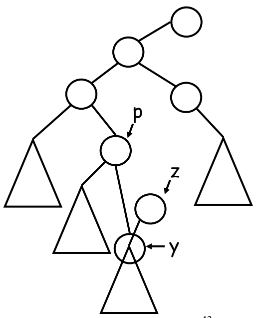
If q.left = null or q.right = null
then z ← q
else z ← min(q.right)
    q.key ← z.key
If z.left ≠ null then y ← z.left
    else y ← z.right
If y ≠ null then
    y.parent ← z.parent
p = y.parent
If z = p.left then p.left = y
    else p.right = y



delete(x,T)

q ← find(x,T)

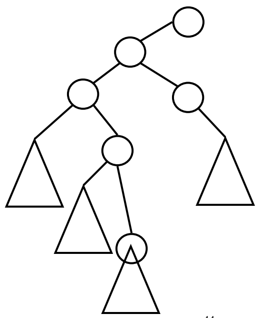
If q.left = null or q.right = null
then z ← q
else z ← min(q.right)
    q.key ← z.key
If z.left ≠ null then y ← z.left
    else y ← z.right
If y ≠ null then
    y.parent ← z.parent
p = y.parent
If z = p.left then p.left = y
    else p.right = y



delete(x,T)

q ← find(x,T)

If q.left = null or q.right = null
then z ← q
else z ← min(q.right)
    q.key ← z.key
If z.left ≠ null then y ← z.left
    else y ← z.right
If y ≠ null then
    y.parent ← z.parent
p = y.parent
If z = p.left then p.left = y
    else p.right = y

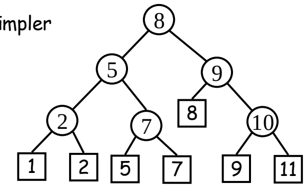


Variation: Items only at the leaves

- Keep elements only at the leaves
- Each internal node contains a number to direct the search

Implementation is simpler (e.g. delete)

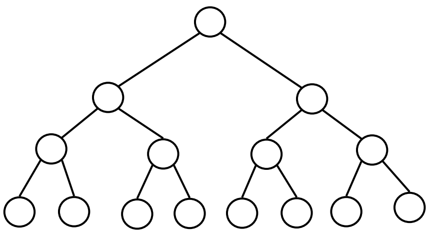
Costs space



Analysis

- Each operation takes O(h) time, where h is the height of the tree
- In general h may be as large as n
- Want to keep the tree with small h

Balance



➔ h = O(log n)

How do we keep the tree balanced through insertions and deletions ?

Applications of search trees

1) Order statistics

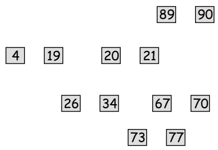
rank and select

Select(i,D)

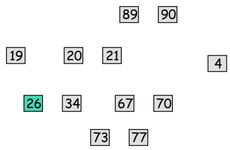
- Select(i,D): Returns the i^{th} element in our predefined set:

An element x such that i-1 elements are smaller than x

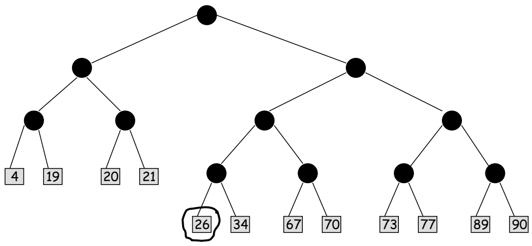
Select(5,D)



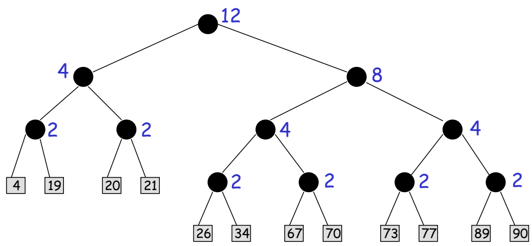
Select(5,D)



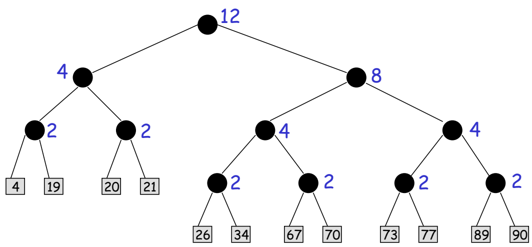
We can use binary trees for this



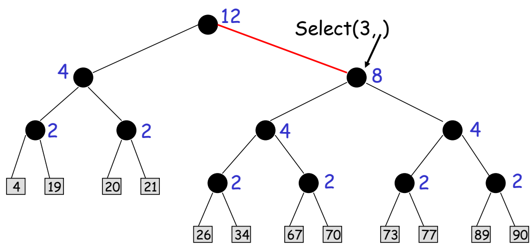
For each node v store # of leaves in the subtree of v



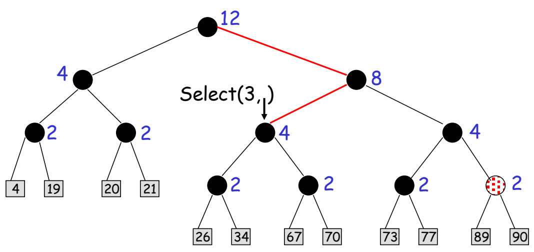
Select(7,T)



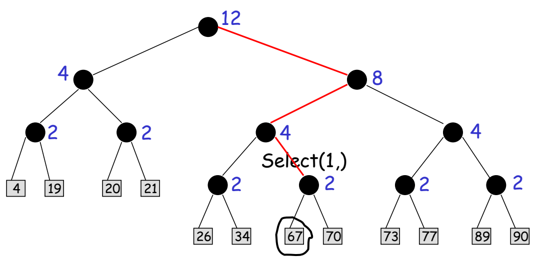
Select(7,T)



Select(7,T)



Select(7,T)

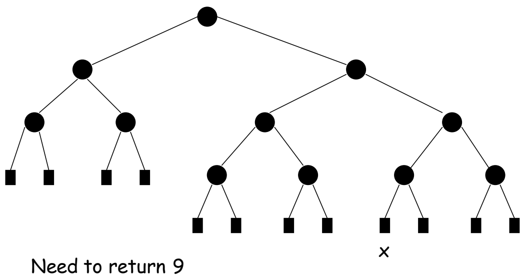


O(logn) worst case time for balanced trees

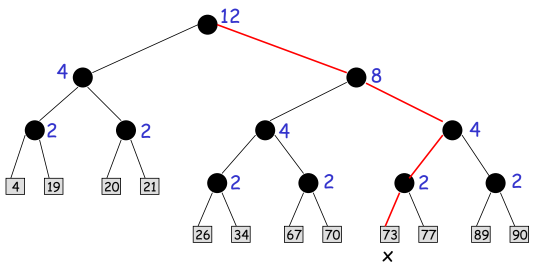
Rank(x,T)

- Return the index of x in T

Rank(x,T)



Need to return 9



Sum up the sizes of the subtrees to the left of the path