# Dictionary

ADT: Provides access

- to elements (via pointers)
- based on keys

Operations

Search : Dict × Key ⟶ Element*

Insert : Dict × Element* ⟶ void

Delete: Dict × Element* ⟶ void

Ordered dictionaries exploit order on key

Min, Max: Dict ⟶ Element*

Pred, Succ : Dict × Element* ⟶ Element*

Implementations:

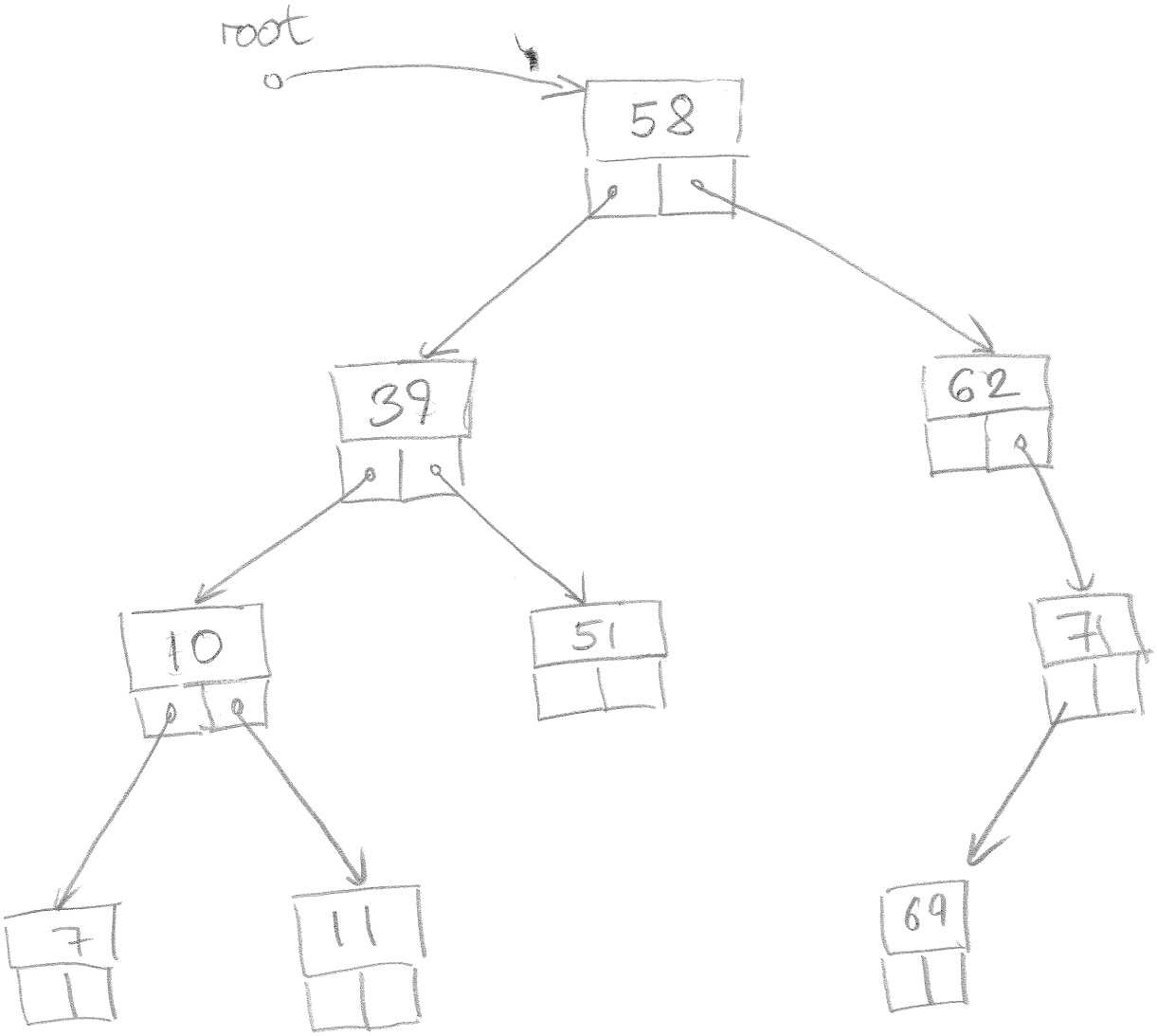
| Operations | Linked List | Ordered List (doubly linked) |
|---|---|---|
| Search | $\Theta(n)$ | $\Theta(n)$ |
| Min, Max | $\Theta(n)$ | $\Theta(1)$ |
| Pred, Succ | $\Theta(n)$ | $\Theta(1)$ |
| Insert | $\Theta(1)$ | $\Theta(n)$ |
| Delete | $\Theta(1)$ | $\Theta(1)$ |

↑ if doubly linked

# Binary Search Trees (BST)

Data structure supporting dictionaries

Fast operations: $\Theta(\lg n)$ for average case




```
class node {
    int key;
    node left;
    node right;
    node parent
}
```

```
class tree {
    node root }
```

# Traversing a Tree

Print keys in order: ITW (= Inorder Tree Walk)

```
ITW(node x)
    if x ≠ nil then
        ITW(x.left)
        print(x.key)
        ITW(x.right)
```

Running time (for balanced trees)

$$T(n) = 2\,T(n/2) + \Theta(1)$$

Master Theorem: $a = 2,\ b = 2,\ c = 0$

$$\Rightarrow T(n) = \Theta(n)$$

What if T is not balanced?

## Searching

```
Search (node x, key k)
    if  x = nil  then  return nil
    if  k = x.key  then  return x
    if  k < x.key
        then return Search(x.left, k)
        else return Search(x.right,
```

Iterative

```
Search (tree T)
    x := T.root
    while x ≠ nil  and  k ≠ x.key do
        if k < x.key
            then x := x.left
            else x := x.right
    return x
```

Search :: Worst-case running time

$\Theta(\text{height}(T))$

height(T) = # edges on longest path in T from root

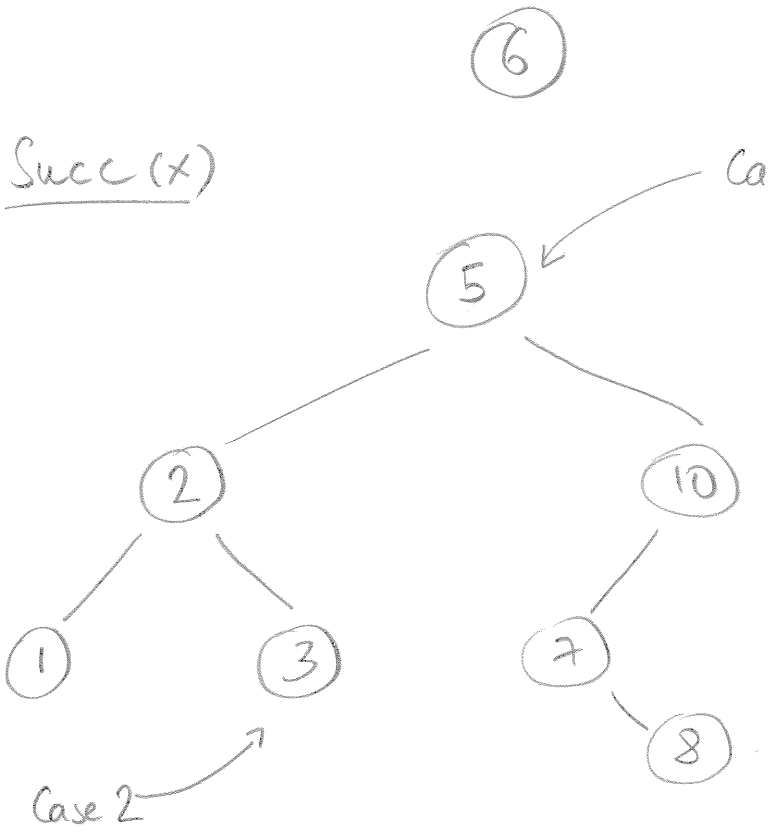
Possibly, height(T) = # nodes(T) − 1

Minimum

```
Min(T)                          look ahead
    x := T.root
    while x.left ≠ nil do
        x := x.left
    return x
```

Running time: $\Theta(\text{height}(T))$

Succ(x)




Case 1: x has right child

$\Rightarrow$ Succ(x) = Min(x.right)

Case 2: x has no right child

$\Rightarrow$ Succ(x) = lowest ancestor z of x
sth x in left subtree of z

$\Rightarrow$ Walk up until you can step right

Successor with trailing pointer

Idea: Introduce $yp := y.parent$ to avoid dereferencing $y.parent$

```
Succ (x)
    if x.right ≠ nil
       then return Min (x.right)
    y := x
    yp := y.parent
    while yp ≠ nil and
                y = yp.right do
          y := yp
          yp := y.parent
    return yp
```

```
Succ(x)
    if x.right ≠ nil
        then return Min(x.right)
    y := x
    while y.parent ≠ nil and
                y = y.parent.right do
        y := y.parent
    return y.parent
```

Idea for Case 2:

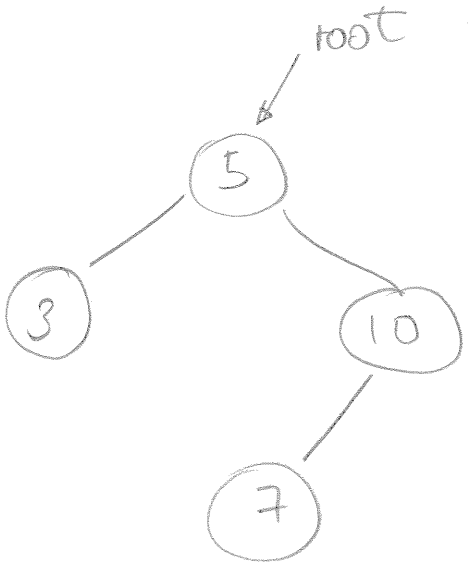
We walk up to the left as long as possible (i.e., the current node y is the right child of its parent)

When we stop,

- the current node is nil (no successor exists) or
- the current node y has a left child (and x is the maximum of y's left subtree)

9

## Insertion




Insert 8

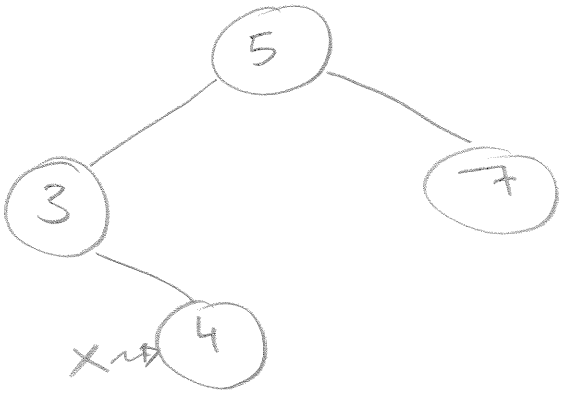
Technique:
trailing pointer

```
Insert (tree T, node x)
  front := T.root; rear := nil
  while front ≠ nil do
      rear := front
      if x.key < front.key
         then front := front.left
         else front := front.right
  if rear = nil
     then x.parent := nil; T.root := x
  elsif x.key < rear.key
     then rear.left := x
     else rear.right := x
  x.parent := rear;
```

Deletion : Delete ( T, x )

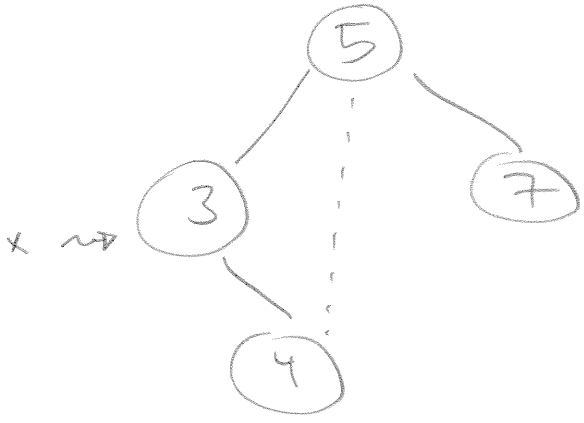
Idea :

Case 1 : x has no children



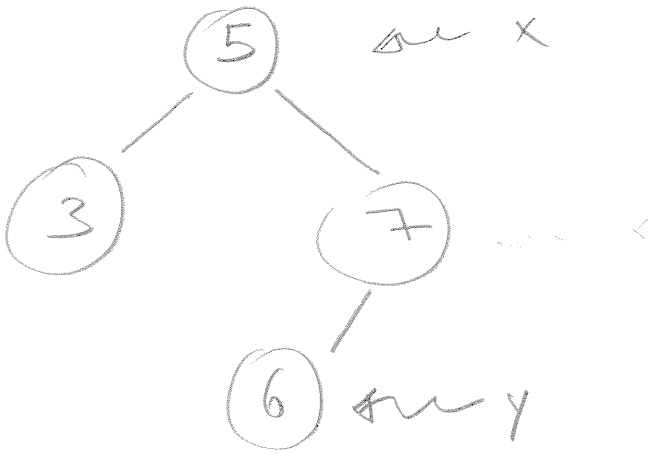

⇒ remove x

Case 2 : x has one child




⇒ splice out x

Case 3: x has two children




⇒ replace x with y := Succ(x);

delete y (y has no left child)

```
Delete (T, x)
    if  x.left = nil  or  x.right = nil
        then  drop := x
        else  drop := Succ(x)
    if  drop.left ≠ nil
        then  keep := drop.left
        else  keep := drop.right
     if  keep ≠ nil
         then  keep.parent := drop.parent
     if  drop.parent = nil
         then  T.root := keep
         else      if  drop = drop.parent.left
                   then  drop.parent.left := keep
                   else  drop.parent.right := keep

      if  drop ≠ x
          then  x.key := drop.key
                % x.info := drop.info
```