Efficient Data Structures

Data Structures and Algorithms (094224)

Yuval Emek

Winter 2022/23

Data structures revisited

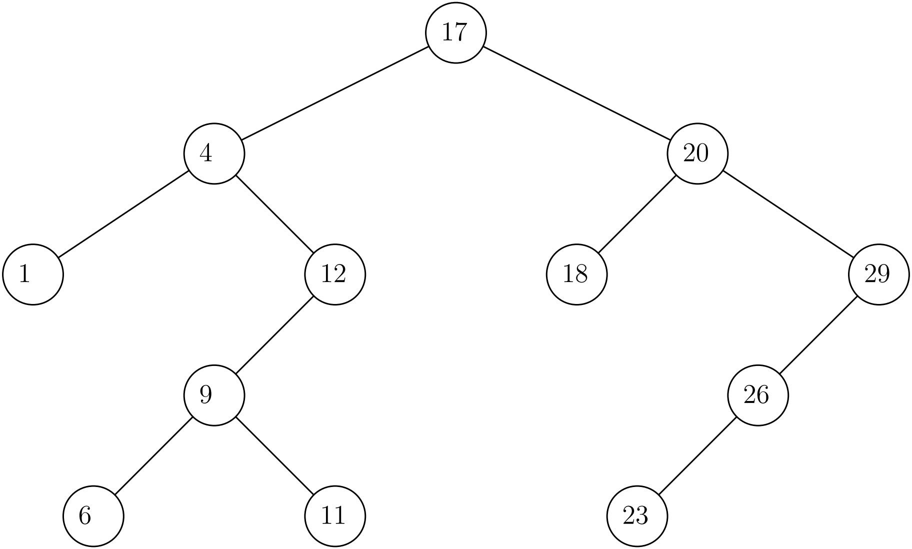
- Focus: a dynamic set of objects
- Objects are typically organized in some well structured digraph
- A designated node attribute: key = the object's identifier
  - Keys are totally ordered
  - For simplicity, assume uniqueness
- Supporting (a subset of the) operations:
  insert, delete, search, minimum, maximum, successor, predecessor
  - FAST!!!
- Actual data stored in satellite attributes
  - Sometimes a pointer to the actual data (storage considerations)

A (rooted) binary tree organization

Binary search tree ⟨עץ חיפוש בינארי⟩

- Objects associated with nodes of a rooted tree T with degrees ≤ 2
  - a.k.a. binary tree
- Data structure's attribute:
  - *root* $=$ pointer to root of T
- Node attributes (additional):
  - *left* $=$ pointer to left child in T
  - *right* $=$ pointer to right child in T
  - p $=$ pointer to parent in T
- The binary search tree property:
  if y is a node in the left (resp., right) subtree of node x, then $y.key < x.key$ (resp., $y.key > x.key$)
  - Binary search tree is not unique for a given set of keys

Example



Searching

Search in the binary search tree rooted at x for a node whose key is k

`Tree_Search`(x, k)

if $x == nil$ or $x.key == k$ **then**
    return x
if $k < x.key$ **then**
    return `Tree_Search`$(x.left, k)$
else return `Tree_Search`$(x.right, k)$

- Run-time: $O(\text{height}(T))$

Finding the minimum/maximum element

Find the node with the smallest key in the binary search tree rooted at x

Tree_Minimum(x)

while $x.left \neq NIL$ **do**
    $x = x.left$
return x

- Undefined if the tree is empty
- Run-time: $O(\mathrm{height}(T))$
- How do we find the maximum?

Finding a successor/predecessor

Find the node y with the smallest key among those with $y.key > x.key$

`Tree_Successor`(x)

if $x.right \neq NIL$ **then**
    return `Tree_Minimum(x.right)`
$y = x.p$
while $y \neq NIL$ and $x == y.right$ **do**
    $x = y$
    $y = y.p$
return y

- Run-time: $O(\mathrm{height}(T))$
- How do we find a predecessor?

the opposite

Inserting a new node

Insert the new node z (DS attributes initialized to NIL) into T

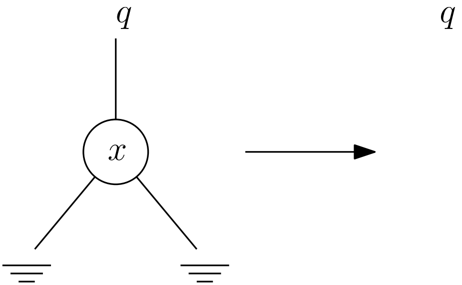
`Tree_Insert`(T, z)

```
if T.root == NIL then
    T.root = z
else
    y = T.root
    x = NIL
    while y ≠ NIL do
        x = y
        if z.key < y.key then
            y = y.left
        else y = y.right
    z.p = x
    if z.key < x.key then
        x.left = z
    else x.right = z
```

- Run-time: $O(\mathrm{height}(T))$

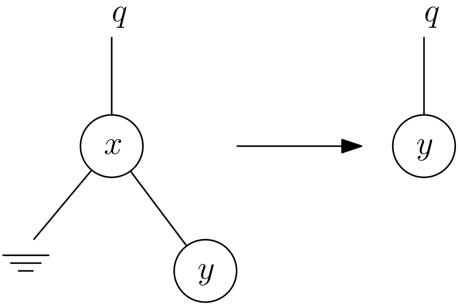
Deleting a node

Deleting node x
case 1: x is a leaf
Not much to do



- Run-time: $O(1)$

Deleting node x
case 2: x has a right child y and no left child
Replace x with y



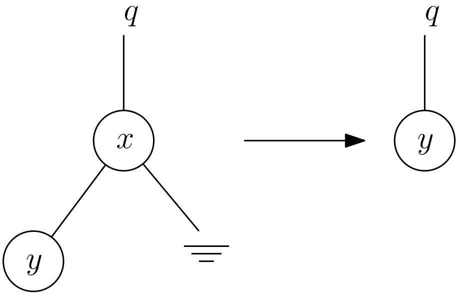
- Run-time: $O(1)$

Deleting a node — cont.

Deleting node x

case 3: x has a left child y and no right child

Replace x with y



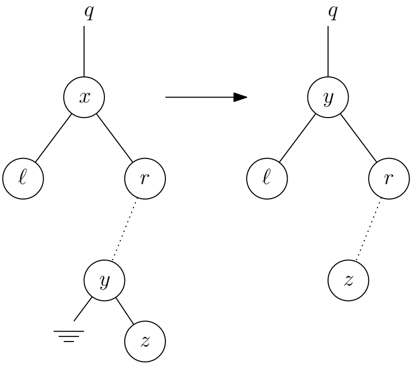
- Run-time: $O(1)$

Deleting a node — cont.

Deleting node x

case 4: x has two children

1. Find x's successor y (in x's right subtree)
2. Swap x and y
3. Remove x
   - We know how to do it because now x has ≤ 1 child



- Run-time: $O(\mathrm{height}(T))$

Balanced trees

- Supported operations:
  insert, delete, search, minimum, maximum, successor, predecessor
- Run-time of each operation in a binary search tree T is $O(\mathrm{height}(T))$
- $\mathrm{height}(T)$ can be $\Omega(n)$ in the worst case
  - How?
  - It is always $\Omega(\log n)$
    - #nodes increases from one level to the next by factor ≤ 2
- Looking for a data structure that supports these operations in time $O(\log n)$ in the worst case
  - Referred to as a *balanced tree* ⟨עץ מאוזן⟩
- There exist balanced tree DSs based on binary search trees
  - AVL trees, red-black trees
- In this lecture: balanced tree DS that follows a different approach
- Note: the height of a binary search tree is $O(\log n)$ on average
  - What does "on average" mean for a DS?
  - Usually efficient in practice

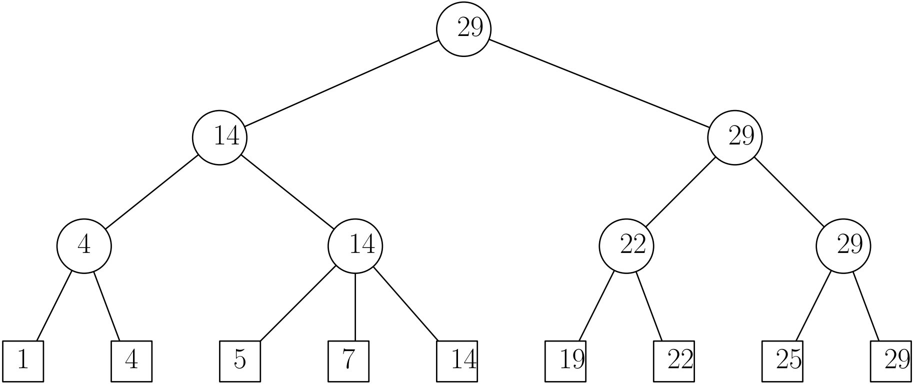
A different kind of rooted tree

2-3 tree ⟨2-3 עץ⟩

- Rooted tree T satisfying:
  - Each internal node has degree 2 or 3
  - All leaves are at the same level
- Objects are stored only at the leaves
  - Internal nodes are there for organization purposes
- Data structure's attribute:
  - *root* = pointer to root of T
- Node attributes:
  - *left* = pointer to left child in T
  - *middle* = pointer to middle child in T
  - *right* = pointer to right child in T
  - *p* = pointer to parent in T
  - *key* = maximum key in its subtree
  - Leaves have additional satellite attributes (storing the actual data)
- When degree is 2, left and middle children are in use
- The 2-3 tree property:

  keys in left subtree $<$ keys in middle subtree $<$ keys in right subtree

Example



Structural properties

- Consider a 2-3 tree T with n leaves
- Observation 1: in the leaf level, keys are sorted from left to right
  - By induction on height
- Observation 2: $\#$nodes in level $i + 1 \geq 2 \times \#$nodes in level i
  - Each internal node has ≥ 2 children
- Corollary 1: $\mathrm{height}(T) = O(\log n)$
  - Will see: run-time of each supported operation is $O(\mathrm{height}(T))$
- Corollary 2: $\#$internal nodes $< \#$leaves
  - By induction on height
  - Don't waste space (asymptotically)

Initialization

Create an (empty) 2-3 tree

`2_3_Init`(T)

new internal node x ▷ DS attributes initialized to *NIL*

new leaves ℓ, m ▷ DS attributes initialized to *NIL*

$\ell.key = -\infty$

$m.key = +\infty$

$\ell.p = m.p = x$

$x.key = +\infty$

$x.left = \ell$

$x.middle = m$

$T.root = x$

- Run-time: $O(1)$
- ℓ and m are called *sentinel nodes*

Searching

Search in the 2-3 tree rooted at x for a node whose key is k

2_3_Search(x, k)

```
1: if x is a leaf then
2:     if x.key == k then
3:         return x
4:     else return NIL
5: if k ≤ x.left.key then
6:     return 2_3_Search(x.left, k)
7: else if k ≤ x.middle.key then
8:     return 2_3_Search(x.middle, k)
9: else return 2_3_Search(x.right, k)
```

- $+\infty$ sentinel node ensures existence of right child in line 9

Finding the minimum/maximum element

Find the leaf with the smallest key in the 2-3 tree T

2_3_Minimum(T)

```
x = T.root
while x is not a leaf do
    x = x.left
x = x.p.middle
if x.key ≠ +∞ then
    return x
else error: T is empty
```

לא חייבים, כי sentinel node

[illegible]

- How do we find the maximum?

חייב להזהר אם ריק..

Finding a successor/predecessor

Find the leaf y with the smallest key among those with $y.key > x.key$

2_3_Successor(x)

```
z = x.p
while x == z.right or (z.right == NIL and x == z.middle) do
    x = z
    z = z.p
if x == z.left then
    y = z.middle
else y = z.right
while y is not a leaf do
    y = y.left
if y.key < +∞ then
    return y
else return NIL
```

כל עוד x הכי ימני נחפש את הנתיב שבו יש x<y

- How do we find a predecessor?

Procedure Update_Key

Update the key of x to the maximum key in its subtree;
(only) $x.middle$ and $x.right$ may be NIL

Update_Key(x)

$x.key = x.left.key$
if $x.middle \neq NIL$ **then**
    $x.key = x.middle.key$
if $x.right \neq NIL$ **then**
    $x.key = x.right.key$

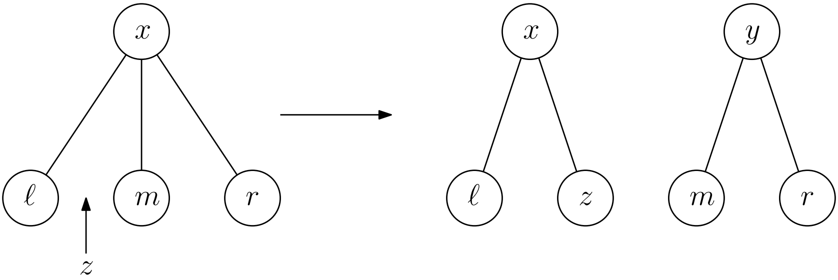
Procedure `Set_Children`

Set ℓ, m, and r to be the left, middle, and right children, respectively, of x; (only) m and r may be NIL

`Set_Children`(x, ℓ, m, r)

$\langle x.left, x.middle, x.right \rangle = \langle \ell, m, r \rangle$
$\ell.p = x$
if $m \neq NIL$ **then**
    $m.p = x$
if $r \neq NIL$ **then**
    $r.p = x$
`Update_Key`(x)

New leaf insertion



New leaf insertion — procedure Insert_And_Split

Insert node z as a child of node x;
split x if necessary and return the new node

Insert_And_Split(x, z)

$\langle \ell, m, r \rangle = \langle x.left, x.middle, x.right \rangle$
if $r == NIL$ **then**
    if $z.key < \ell.key$ **then**
        Set_Children(x, z, ℓ, m)
    else if $z.key < m.key$ **then**
        Set_Children(x, ℓ, z, m)
    else Set_Children(x, ℓ, m, z)
    return NIL
new internal node y
...

New leaf insertion — procedure Insert_And_Split

Insert_And_Split(x, z) — cont.

```
if z.key < ℓ.key then
    Set_Children(x, z, ℓ, NIL)
    Set_Children(y, m, r, NIL)
else if z.key < m.key then
    Set_Children(x, ℓ, z, NIL)
    Set_Children(y, m, r, NIL)
else if z.key < r.key then
    Set_Children(x, ℓ, m, NIL)
    Set_Children(y, z, r, NIL)
else Set_Children(x, ℓ, m, NIL)
    Set_Children(y, r, z, NIL)
return y
```

Inserting a new leaf

Insert the new leaf z (DS attributes initialized to NIL) into T

`2_3_Insert`(T, z)

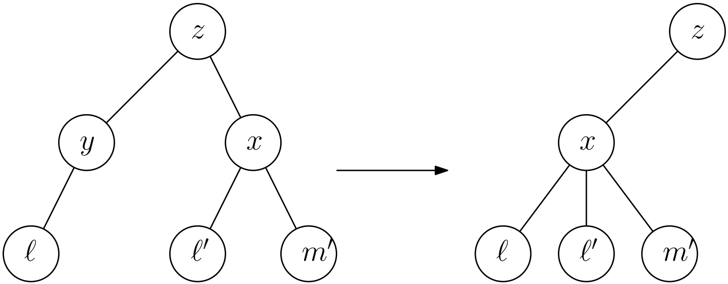
```
y = T.root
while y is not a leaf do
    if z.key < y.left.key then y = y.left
    else if z.key < y.middle.key then y = y.middle
    else y = y.right
x = y.p
z = Insert_And_Split(x, z)
while x ≠ T.root do
    x = x.p
    if z ≠ NIL then
        z = Insert_And_Split(x, z)
    else Update_Key(x)
...
```

Inserting a new leaf

`2_3_Insert`(T, z) — cont.

```
if z ≠ NIL then
    new internal node w
    Set_Children(w, x, z, NIL)
    T.root = w
```

Leaf deletion



Leaf deletion — procedure Borrow_Or_Merge

Borrow a child from a sibling x of y or merge x and y;
return a pointer to the parent of y (and x)

Borrow_Or_Merge(y)

```
z = y.p
if y == z.left then
    x = z.middle
    if x.right ≠ NIL then
        Set_Children(y, y.left, x.left, NIL)
        Set_Children(x, x.middle, x.right, NIL)
    else Set_Children(x, y.left, x.left, x.middle)
        delete y
        Set_Children(z, x, z.right, NIL)
    return z
...
```

Leaf deletion — procedure Borrow_Or_Merge

Borrow_Or_Merge(y) — cont.

```
if y == z.middle then
    x = z.left
    if x.right ≠ NIL then
        Set_Children(y, x.right, y.left, NIL)
        Set_Children(x, x.left, x.middle, NIL)
    else Set_Children(x, x.left, x.middle, y.left)
        delete y
        Set_Children(z, x, z.right, NIL)
    return z
...
```

Leaf deletion — procedure Borrow_Or_Merge

Borrow_Or_Merge(y) — cont.

```
x = z.middle
if x.right ≠ NIL then
    Set_Children(y, x.right, y.left, NIL)
    Set_Children(x, x.left, x.middle, NIL)
else Set_Children(x, x.left, x.middle, y.left)
    delete y
    Set_Children(z, z.left, x, NIL)
return z
```

Deleting a leaf

Delete leaf x from T

`2_3_Delete`(T, x)

```
y = x.p
if x == y.left then
    Set_Children(y, y.middle, y.right, NIL)
else if x == y.middle then
    Set_Children(y, y.left, y.right, NIL)
else Set_Children(y, y.left, y.middle, NIL)
delete x
...                                        ▷ deg(y) may be < 2
```

Deleting a leaf

`2_3_Delete`(T, x) — cont.

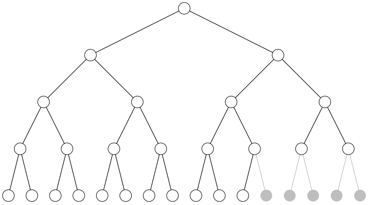
```
while y ≠ NIL do
    if y.middle == NIL then
        if y ≠ T.root then
            y = Borrow_Or_Merge(y)
        else T.root = y.left
            y.left.p = NIL
            delete y
            return
    else Update_Key(y)
        y = y.p
```

More on 2-3 trees

- All operations run in time $O(\mathrm{height}(T)) = O(\log n)$
  - A balanced tree DS
- Sometimes the actual keys are stored only at the leaves
  - Internal nodes store pointers to keys in the corresponding leaves
  - Saving storage if keys themselves are large
- 2-3 trees are a special case of B^+ trees
  - Parameterized by a degree parameter d
  - The degree of each internal node x satisfies $\lceil d/2 \rceil \leq \deg(x) \leq d$
  - The degree of the root r satisfies $2 \leq \deg(r) \leq d$
  - Height proportional to $\log_d n$
- Often: x stores maximum key in each child subtree
  - $\deg(x)$ keys in total
- Very efficient for external memories
  - Access is very expensive in comparison to internal memory
  - Read/write a whole page rather than a single address
  - Adjust the parameter d so that size of a node $\approx$ size of page
    - Save on #page read/write operations

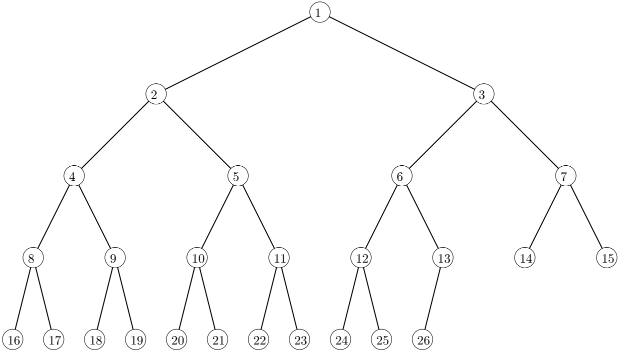
Binary trees revisited

- A *complete* ⟨שלם⟩ binary tree:
  - the degree of each internal node is exactly 2; and
  - all leaves are at the same depth
- Height h complete binary tree has 2^h leaves and $2^h - 1$ internal nodes
- A *nearly complete* ⟨כמעט שלם⟩ binary tree:
  - obtained from a height h complete binary tree by deleting the k rightmost leaves for some $0 \leq k < 2^h$
- Properties of nearly complete binary trees:
  - A generalization of complete binary trees
  - All leaves are in the last two levels
  - $\#$internal nodes $\leq \#$leaves $\leq \#$internal nodes $+ 1$



Array representation

- Representing an n-node nearly complete binary tree in array $A[1 \dots n]$
  - The root is identified with $A[1]$
  - Left child of internal node $A[i]$ is identified with $A[2i]$
  - Right child of internal node $A[i]$ is identified with $A[2i+1]$
  - Parent of non-root node $A[i]$ is identified with $A[\lfloor i/2 \rfloor]$
  - Define $\texttt{Left}(i) = 2i$, $\texttt{Right}(i) = 2i + 1$, $\texttt{Parent}(i) = \lfloor i/2 \rfloor$



A nearly complete binary tree data structure

Binary heap ⟨ערמה בינארית⟩
(sometimes simply heap)

- Objects associated with the nodes of a nearly complete binary tree T
- T represented in (prefix of) array $A[1 \ldots A.length]$
  - $A.length$ is an upper bound on #objects in the dynamic set
    - Known in advance
    - Exists in many applications
  - n objects stored in $A[1 \ldots n]$
- Data structure's attribute (on top of the standard array's attributes):
  - $heap\text{-}size = \#$currently stored objects ($n \leq A.length$)
- The binary heap property:
  $A[\mathtt{Parent}(i)].key < A[i].key$ for every non-root $A[i]$ each almost complete binary tree that have this is a binary heap
- Sometimes referred to as a *priority queue* ⟨תור עדיפויות⟩ or *minimum heap* ⟨ערמת מינימום⟩
  - Switching the relevant operations/inequalities yields a *maximum heap* ⟨ערמת מקסימום⟩

Building a heap — Procedure Heapify

Precondition: subtrees rooted at $A[\texttt{Left}(i)]$ and $A[\texttt{Right}(i)]$ are heaps; modify A so that the subtree rooted at $A[i]$ is a heap

$\texttt{Heapify}(A, i)$

$\ell = \texttt{Left}(i)$
if $\ell \leq A.\mathit{heap\text{-}size}$ and $A[\ell].\mathit{key} < A[i].\mathit{key}$ **then**
    $\mathit{smallest} = \ell$
else $\mathit{smallest} = i$
$r = \texttt{Right}(i)$
if $r \leq A.\mathit{heap\text{-}size}$ and $A[r].\mathit{key} < A[\mathit{smallest}].\mathit{key}$ **then**
    $\mathit{smallest} = r$
if $\mathit{smallest} \neq i$ **then**
    swap $A[i]$ and $A[\mathit{smallest}]$
    $\texttt{Heapify}(A, \mathit{smallest})$

- $A[i]$ seeps down the tree
- Run-time: $O(\mathrm{height}(T_i)) = O(\log n)$
  - T_i = subtree rooted at $A[i]$

Building a heap from an arbitrary array

Build a heap from the objects in $A[1 \ldots A.length]$

`Build_Heap(A)`

1: $A.heap\text{-}size = A.length$
2: **for** $i = A.length, \ldots, 2, 1$ **do**
3:     `Heapify(A, i)`

- Naive run-time analysis:
  - n iterations
  - $O(\log n)$ time per iteration
  - $\Longrightarrow O(n \log n)$
- Can do better

Building a heap — run-time analysis

- Recall: run-time of $\texttt{Heapify}(A, i)$ is $O(\mathrm{height}(T_i))$
- Height of a nearly complete binary tree T with n nodes is $\lfloor \lg n \rfloor$
- $\#$nodes of height j in $T \leq 2^{\lfloor \lg n \rfloor - j} = O\left(n/2^j\right)$
- $\Longrightarrow$ run-time of $\texttt{Build\_Heap}$ is

$$O\left(\sum_{j=0}^{\lfloor \lg n \rfloor} j \cdot \frac{n}{2^j}\right) \leq O(n) \cdot \sum_{j=0}^{\infty} \frac{j}{2^j} = O(n)$$

Extracting the minimum

Delete the minimum object from heap A and return it

Heap_Extract_Min(A)

if $A.\mathit{heap\text{-}size} < 1$ **then**
    error "the heap is empty"
$\mathit{min} = A[1]$
$A[1] = A[A.\mathit{heap\text{-}size}]$
$A.\mathit{heap\text{-}size} = A.\mathit{heap\text{-}size} - 1$
$\mathrm{Heapify}(A, 1)$
return min

- Run-time: $O(\log n)$
- What's the run-time of returning (a pointer to) the minimum object without deleting it? O(1)

Decreasing a key

Precondition: $k < A[i].key$;
change the key of $A[i]$ to k
$\texttt{Heap\_Decrease\_Key}(A, i, k)$

```
if k > A[i].key then
    error "new key is larger than current key"
A[i].key = k
while i > 1 and A[i].key < A[Parent(i)].key do
    swap A[i] and A[Parent(i)]
    i = Parent(i)
```

- The updated $A[i]$ seeps up the tree
- Run-time: $O(\log n)$

Inserting a new node

Insert the new heap node x into heap A

Heap_Insert(A, x)

$s = A.\mathit{heap\text{-}size} + 1$

$A[s] = x$ ▷ copy all satellite attributes

$A[s].\mathit{key} = \infty$

$A.\mathit{heap\text{-}size} = s$ ▷ a valid heap of size s

Heap_Decrease_Key$(A, s, x.\mathit{key})$

- Run-time: $O(\log n)$

Comparison to balanced trees

- Balanced trees can be adapted to support:
  - Returning a pointer to the minimum object in time $O(1)$
  - Extracting the minimum object in time $O(\log n)$
  - Decreasing the key of a given node in time $O(\log n)$
  - How?
- Advantages of heaps over balanced trees:
  - Building a new heap with n objects is (asymptotically) faster
  - Speed-up (non-asymptotic) due to smaller constants
  - Speed-up (non-asymptotic) due to array representation
    - No pointers, whole DS stored in the same memory page