B-trees, Heaps and Heapsort

Congduan Li

Chinese University of Hong Kong, Shenzhen

congduan.li@gmail.com

Oct 17 & 19, 2017

How do we aug. data structures

1. Choose an underlying data structure.
2. Determine additional information to maintain.
3. Verify that we can maintain additional information for existing data structure operations.
4. Develop new operations.

Don't need to do these steps in strict order! Usually do a little of each, in parallel.

Don't need to do these steps in strict order! Usually do a little of each, in parallel.

How did we do them for OS trees?

1. R-B tree.
2. $size[x]$.
3. Showed how to maintain *size* during insert and delete.
4. Developed OS-SELECT and OS-RANK.

R-B trees are amenable

Red-black trees are particularly amenable to augmentation.

Theorem

Augment a R-B tree with field f, where $f[x]$ depends only on information in x, $left[x]$, and $right[x]$ (including $f[left[x]]$ and $f[right[x]]$). Then can maintain values of f in all nodes during insert and delete without affecting $O(\lg n)$ performance.

Proof Since $f[x]$ depends only on x and its children, when we alter information in x, changes propagate only upward (to $p[x]$, $p[p[x]]$, ..., $root$).

Height = $O(\lg n) \Rightarrow O(\lg n)$ updates, at $O(1)$ each.

R-B trees are amenable

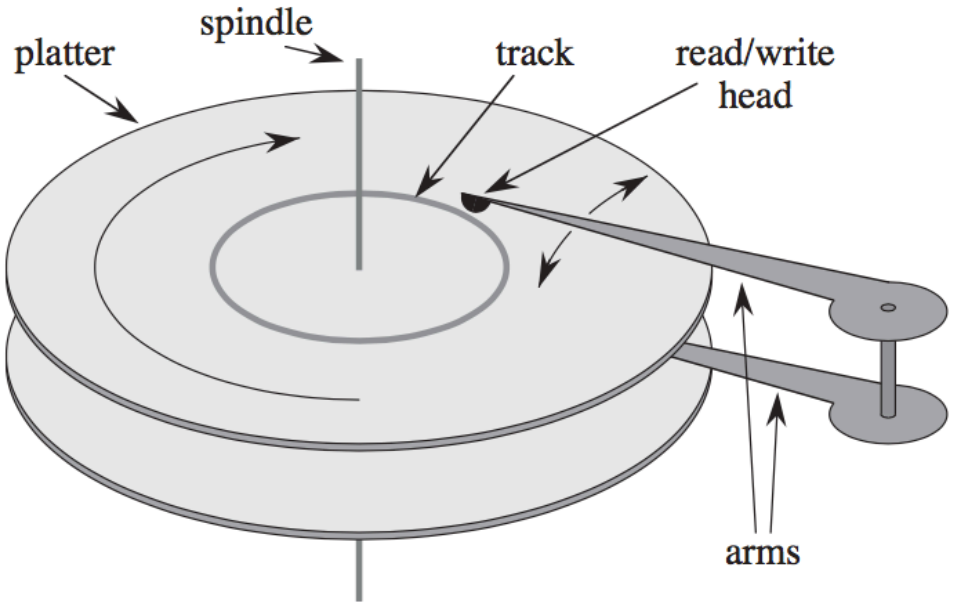
Insertion: Insert a node as child of existing node. Even if can't update f on way down, can go up from inserted node to update f. During fixup, only changes come from color changes (no effect on f) and rotations. Each rotation affects f of ≤ 3 nodes (x,y, and parent), and can recompute each in $O(1)$ time. Then, if necessary, propagate changes up the tree. Therefore, $O(\lg n)$ time per rotation. Since ≤ 2 rotations, $O(\lg n)$ time to update f during fixup.

Delete: Same idea. After splicing out a node, go up from there to update f. Fixup has ≤ 3 rotations. $O(\lg n)$ per rotation $\Rightarrow O(\lg n)$ to update f during fixup.

■ (theorem)

For some attributes, can get away with $O(1)$ per rotation. Example: *size* field.

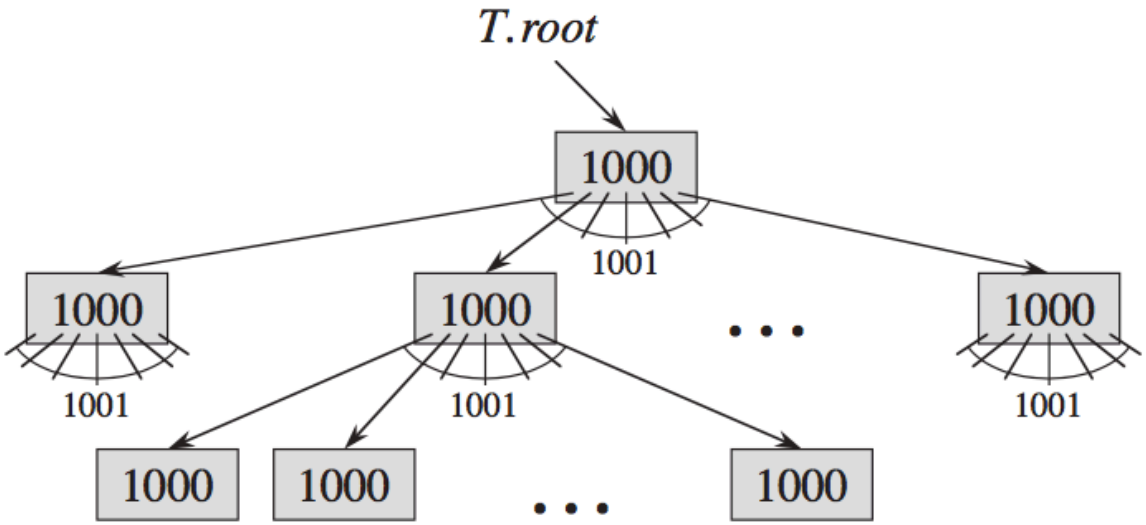
B trees (Chap 18)



- B tree: balanced tree working well for secondary storages, e.g, disks
- Time consuming: disk read
- Basic unit for read: page, fixed length of bits
- Try to make each page a node to reduce disk reads

disk → memory
1 node

B trees example

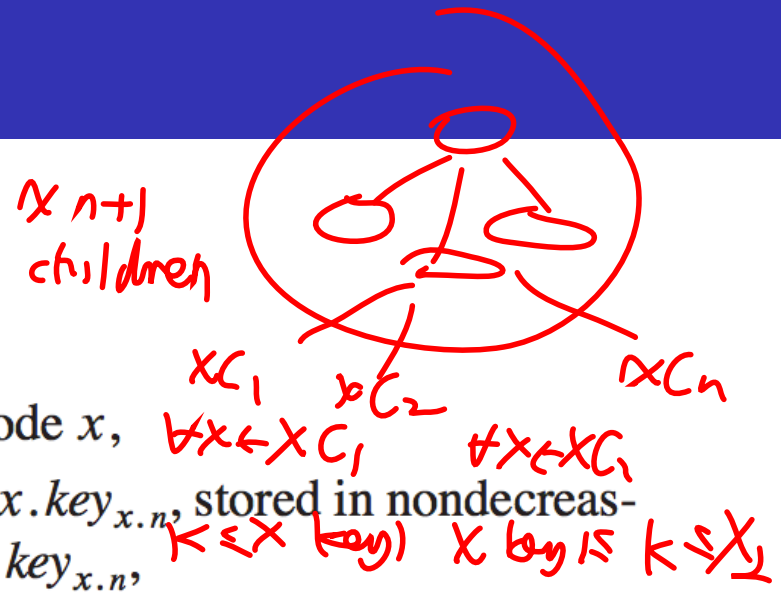


1 node,
1000 keys

1001 nodes,
1,001,000 keys

1,002,001 nodes,
1,002,001,000 keys

B trees definition



1. Every node x has the following attributes:
   a. $x.n$, the number of keys currently stored in node x,
   b. the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \leq x.key_2 \leq \cdots \leq x.key_{x.n}$,
   c. $x.leaf$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains $x.n + 1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.
3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots \leq x.key_{x.n} \leq k_{x.n+1} .$$

B trees definition

4. All leaves have the same depth, which is the tree's height h.
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the ***minimum degree*** of the B-tree:

   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

   b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t - 1$ keys.[2]

$t-1 \leq x.n \leq 2t-1$ $t \geq 2$

for $t=2$ $1 \leq x.n \leq 3$

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and we have a ***2-3-4 tree***. In practice, however, much larger values of t yield B-trees with smaller height.

$\log_t n$, $t \uparrow$ $h \downarrow$

B trees height

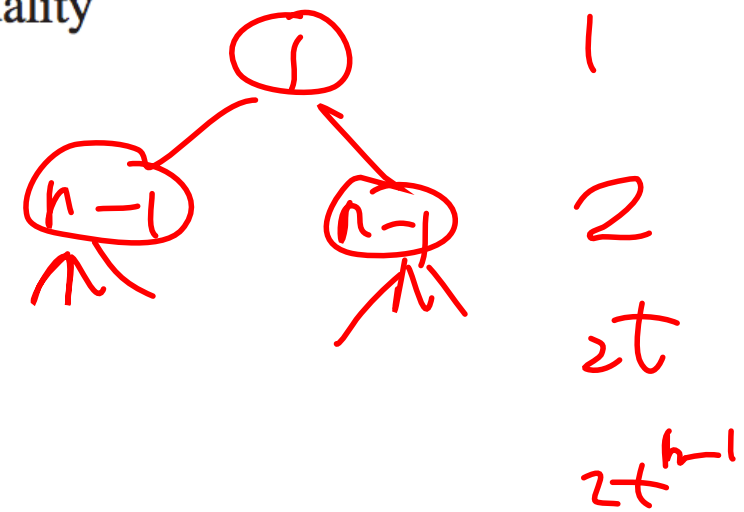
Theorem 18.1

If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2} .$$

Proof The root of a B-tree T contains at least one key, and all other nodes contain at least $t-1$ keys. Thus, T, whose height is h, has at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth h it has at least $2t^{h-1}$ nodes. Figure 18.4 illustrates such a tree for $h = 3$. Thus, the number n of keys satisfies the inequality

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 . \end{aligned}$$



By simple algebra, we get $t^h \leq (n+1)/2$. Taking base-t logarithms of both sides proves the theorem. ■

$h \leq \log_t \frac{n+1}{2} = \log_t (n+1) - \log_t 2$

B trees search

```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3       i = i + 1
4  if i ≤ x.n and k == x.key_i
5       return (x, i)
6  elseif x.leaf
7       return NIL
8  else DISK-READ(x.c_i)
9       return B-TREE-SEARCH(x.c_i, k)
```

Running time: $O(th) = O(t \log_t n)$

worst case
all nodes

B trees search example

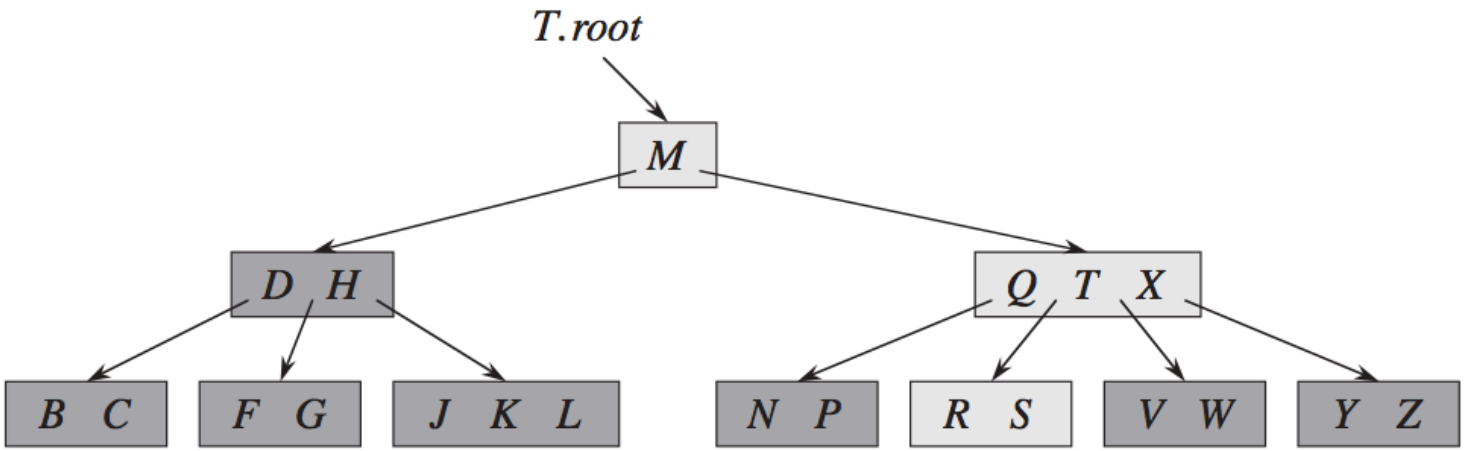


Figure 18.1 illustrates the operation of B-TREE-SEARCH. The procedure examines the lightly shaded nodes during a search for the key R.

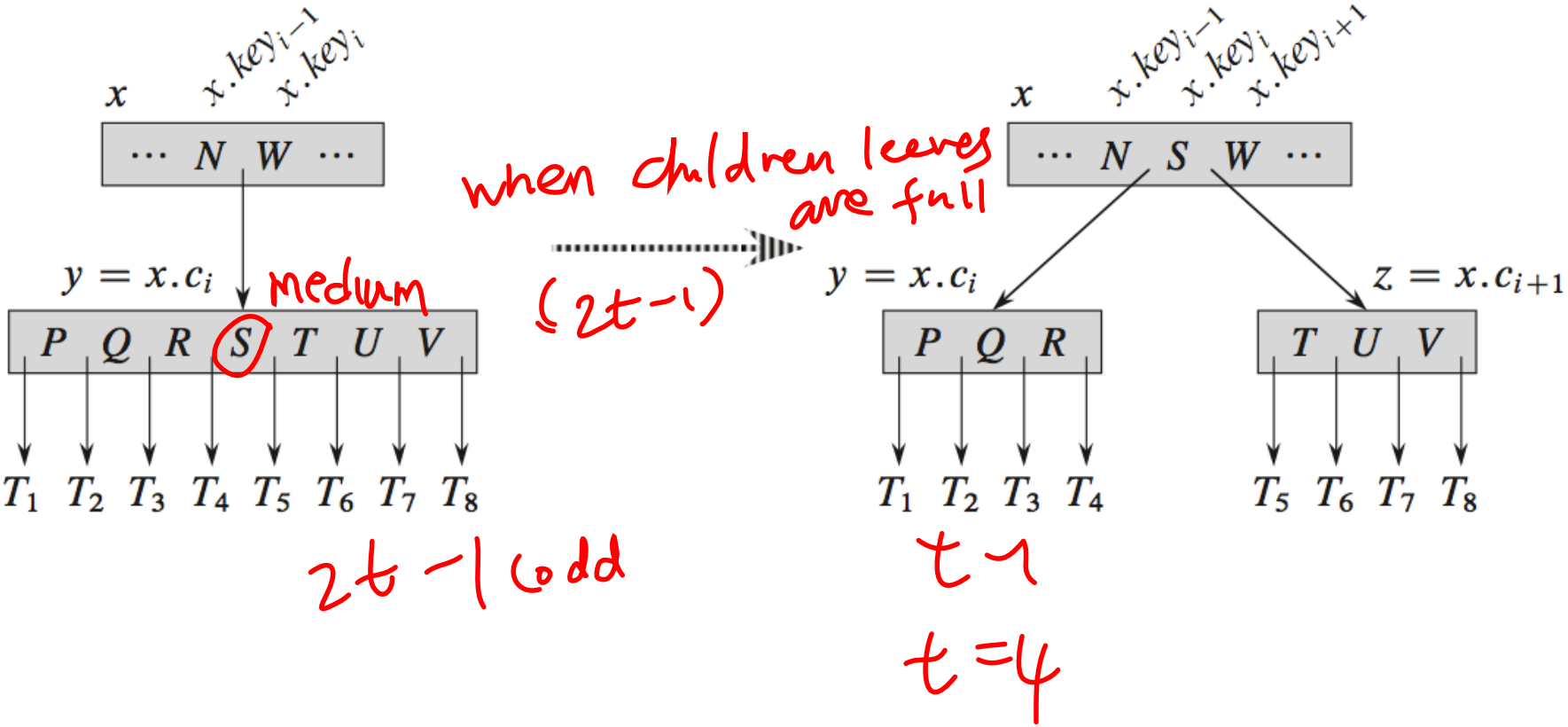
Create a B tree

B-TREE-CREATE(T)

1 x = ALLOCATE-NODE() find space in disk for x
2 $x.leaf$ = TRUE
3 $x.n = 0$
4 DISK-WRITE(x)
5 $T.root = x$

B-TREE-CREATE requires $O(1)$ disk operations and $O(1)$ CPU time.

Split child


x
$x.key_{i-1}$
$x.key_i$
… N W …
$y = x.c_i$
medium
P Q R S T U V
T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8
2t-1 (odd
when children leaves are full
(2t-1)
x
$x.key_{i-1}$
$x.key_i$
$x.key_{i+1}$
… N S W …
$y = x.c_i$
$z = x.c_{i+1}$
P Q R
T U V
T_1 T_2 T_3 T_4
T_5 T_6 T_7 T_8
t-1
t=4

Split child: $O(1)$

$O(t)$ t fixed

B-TREE-SPLIT-CHILD(x, i)

```
1  z = ALLOCATE-NODE()
2  y = x.c_i
3  z.leaf = y.leaf
4  z.n = t − 1
5  for j = 1 to t − 1
6      z.key_j = y.key_{j+t}
7  if not y.leaf
8      for j = 1 to t
9          z.c_j = y.c_{j+t}
10 y.n = t − 1
11 for j = x.n + 1 downto i + 1
12     x.c_{j+1} = x.c_j
13 x.c_{i+1} = z
14 for j = x.n downto i
15     x.key_{j+1} = x.key_j
16 x.key_i = y.key_t
17 x.n = x.n + 1
18 DISK-WRITE(y)
19 DISK-WRITE(z)
20 DISK-WRITE(x)
```

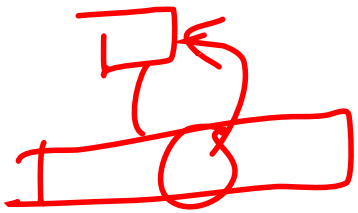
change children pointer

B trees insert

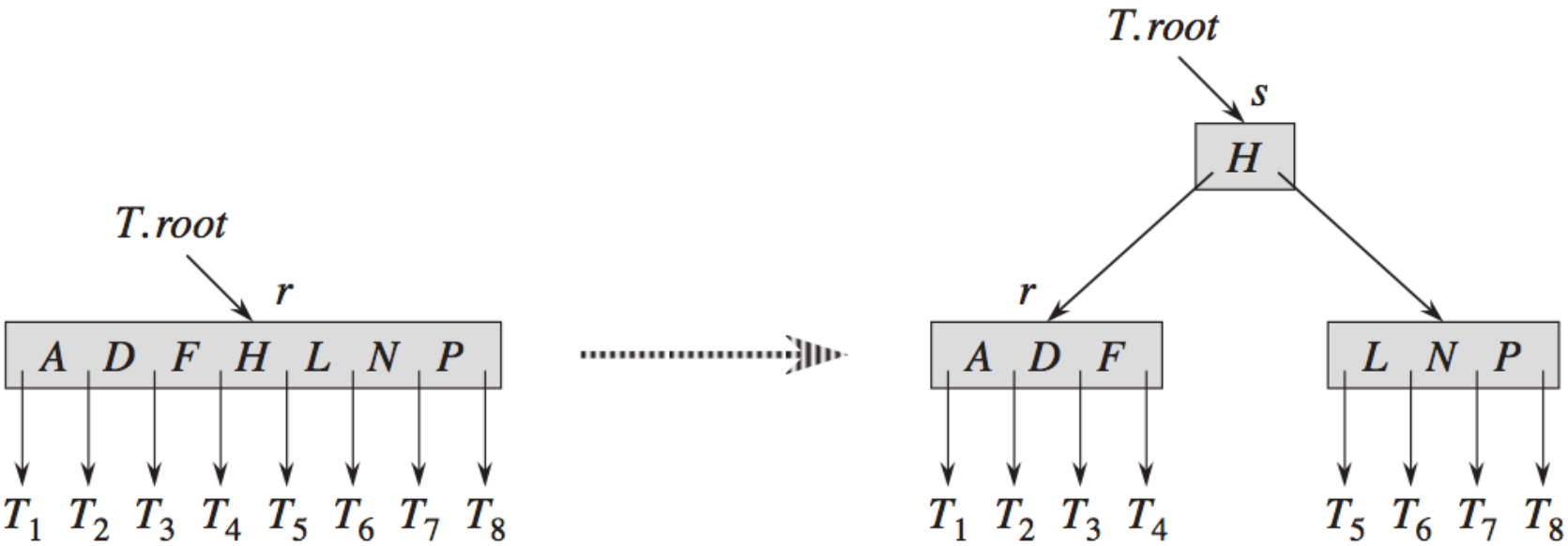
```
B-TREE-INSERT(T, k)
 1  r = T.root
 2  if r.n == 2t − 1
 3      s = ALLOCATE-NODE()
 4      T.root = s
 5      s.leaf = FALSE
 6      s.n = 0
 7      s.c_1 = r
 8      B-TREE-SPLIT-CHILD(s, 1)
 9      B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

root is full

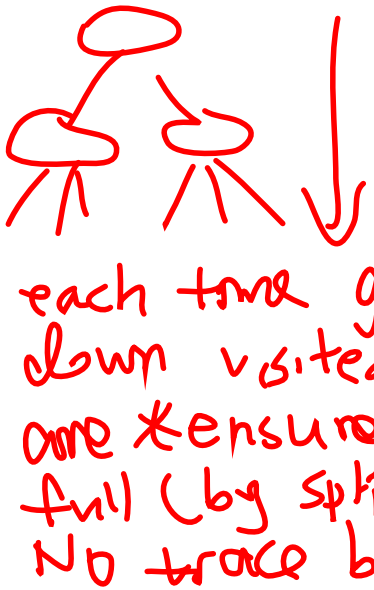
creat new root



Split root


T.root
r
A D F H L N P
T1 T2 T3 T4 T5 T6 T7 T8
T.root
s
H
r
A D F
L N P
T1 T2 T3 T4
T5 T6 T7 T8

B trees insert nonfull



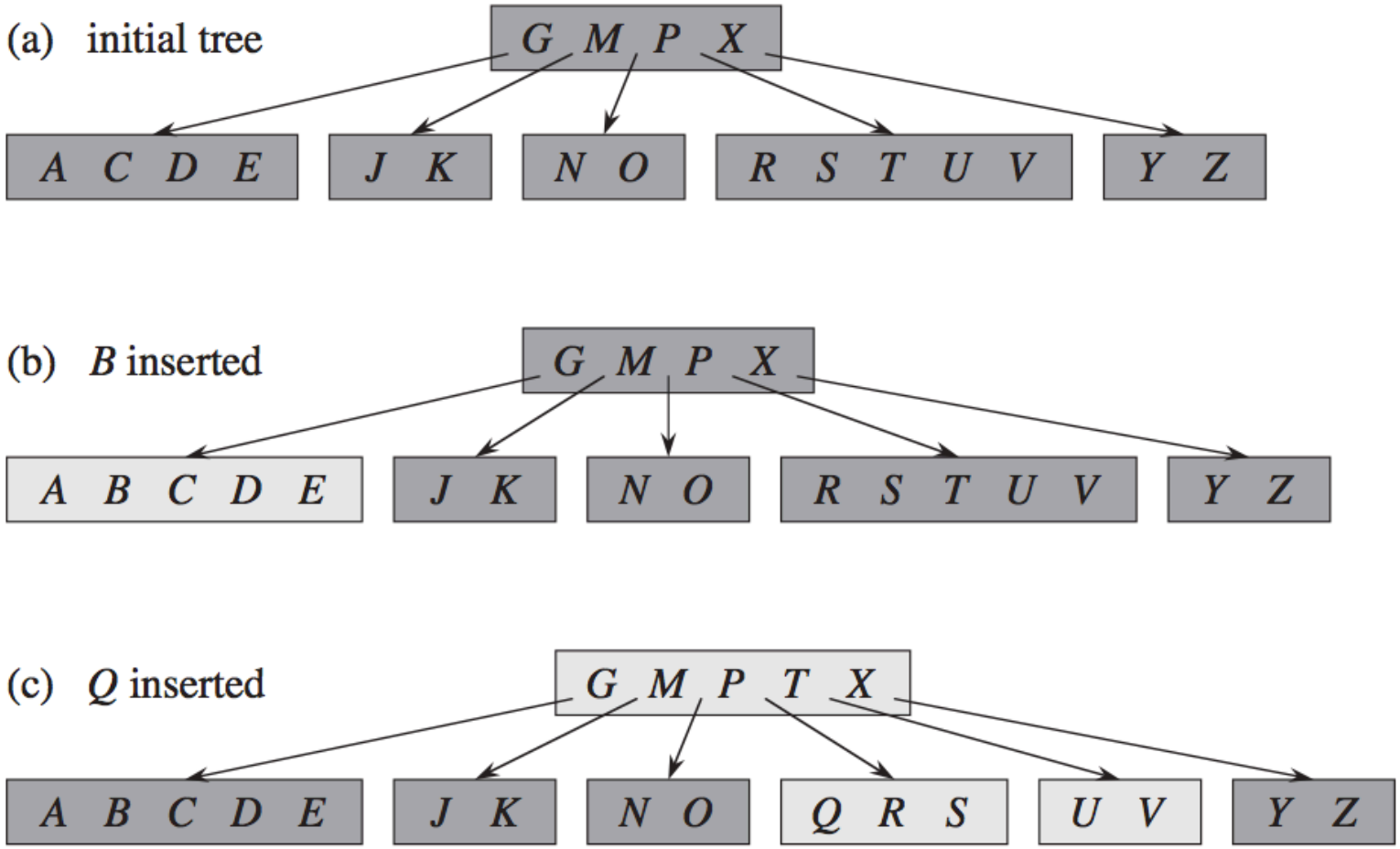
do not need to read from disk again

```
B-TREE-INSERT-NONFULL(x, k)
 1  i = x.n
 2  if x.leaf
 3      while i ≥ 1 and k < x.key_i
 4          x.key_{i+1} = x.key_i
 5          i = i − 1
 6      x.key_{i+1} = k
 7      x.n = x.n + 1
 8      DISK-WRITE(x)
 9  else while i ≥ 1 and k < x.key_i
10          i = i − 1
11      i = i + 1
12      DISK-READ(x.c_i)
13      if x.c_i.n == 2t − 1
14          B-TREE-SPLIT-CHILD(x, i)
15          if k > x.key_i
16              i = i + 1
17      B-TREE-INSERT-NONFULL(x.c_i, k)
```

Running time: $O(th) = O(t \log_t n)$

B trees insertion examples

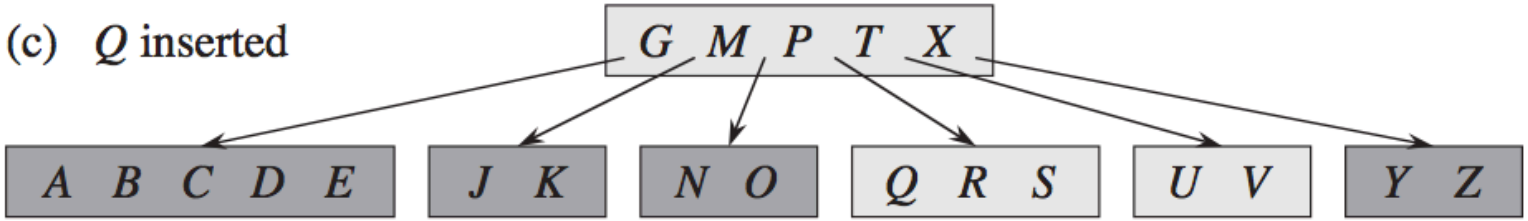
t=3



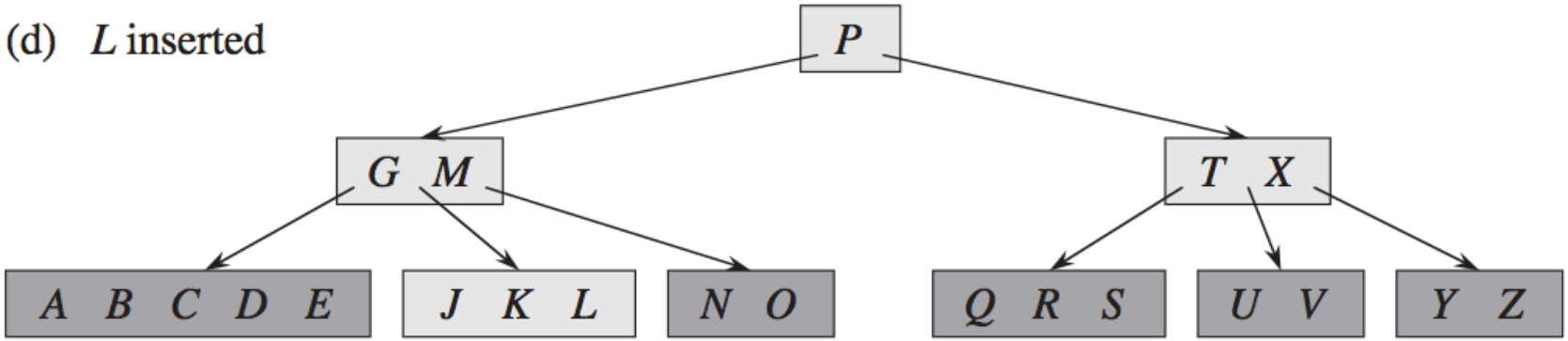
Split does not affect the height except the root
Maintain b tree

B trees insertion examples

(c) Q inserted

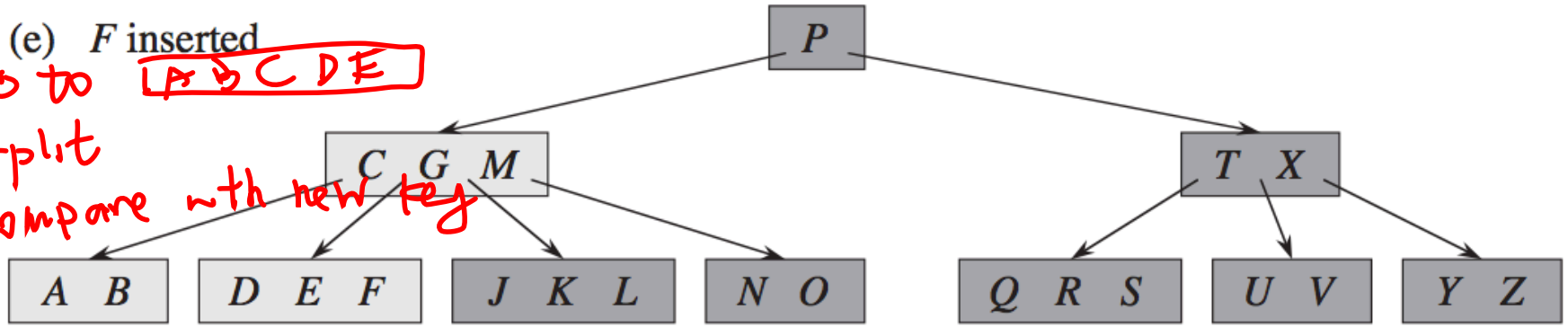


(d) L inserted



(e) F inserted

go to A B C D E
split
Compare with new key

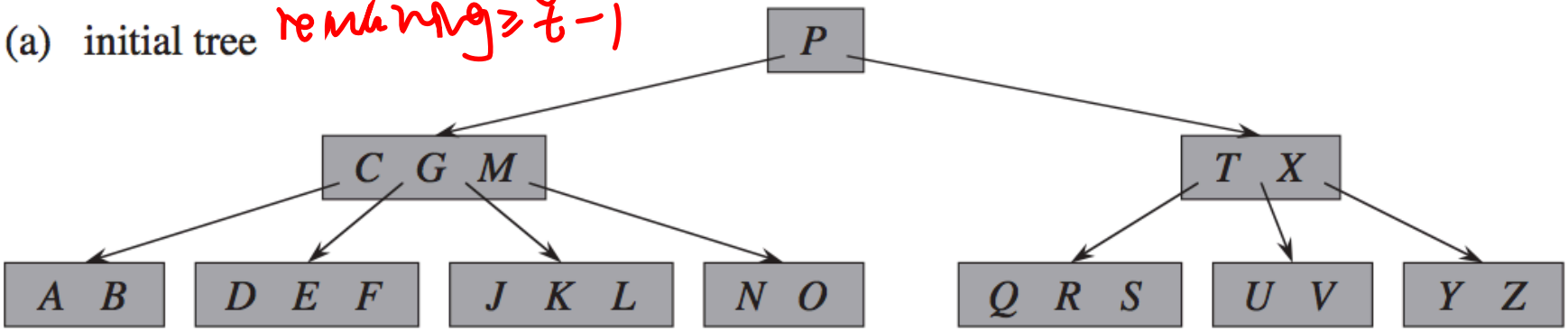


B trees deletion

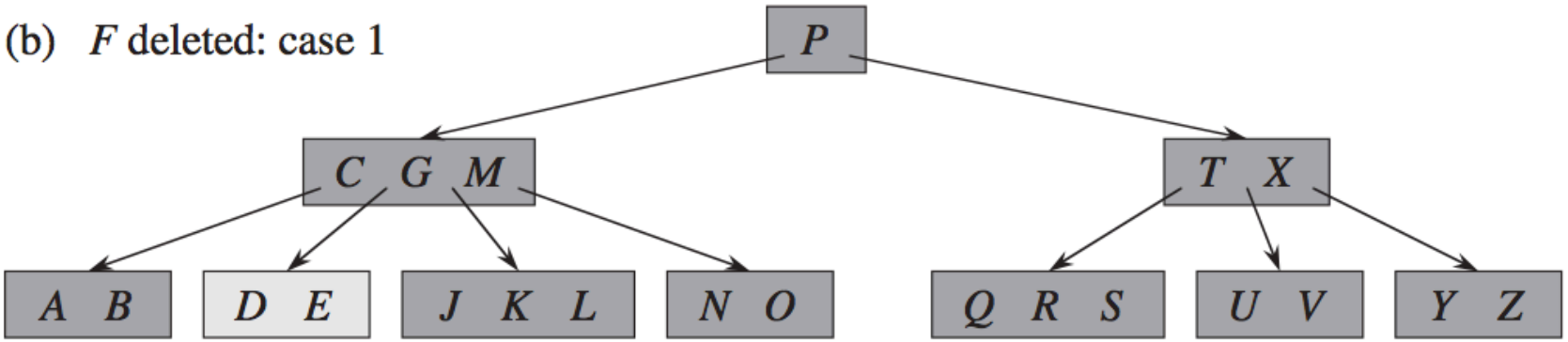
1. If the key k is in node x and x is a leaf, delete the key k from x.

node has $\geq t$ keys delete
remaining $\geq t-1$

(a) initial tree



(b) F deleted: case 1



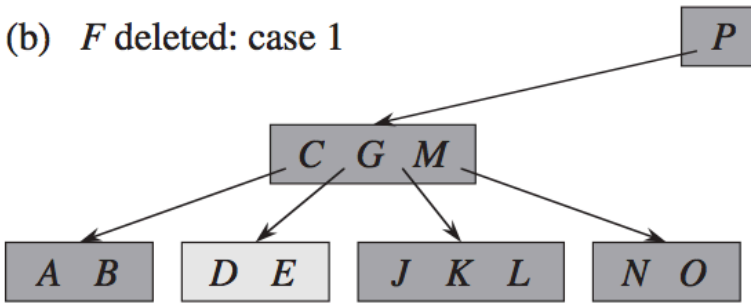
B trees deletion

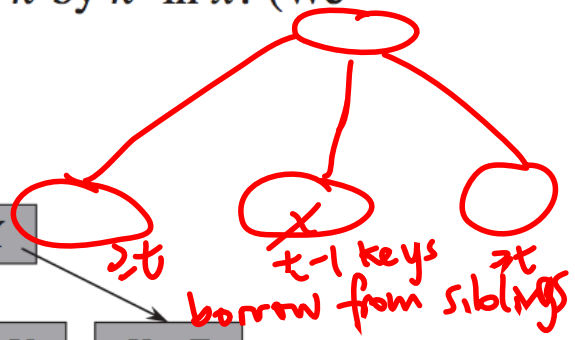
2. If the key k is in node x and x is an internal node, do the following:

a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

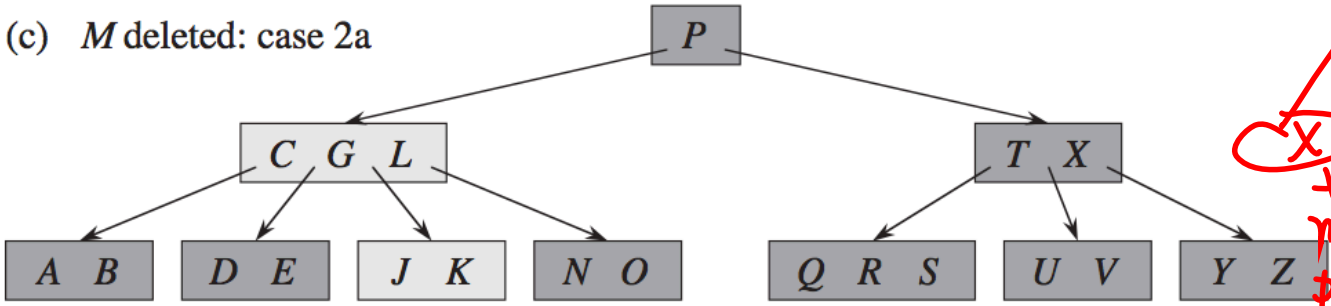
b. If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

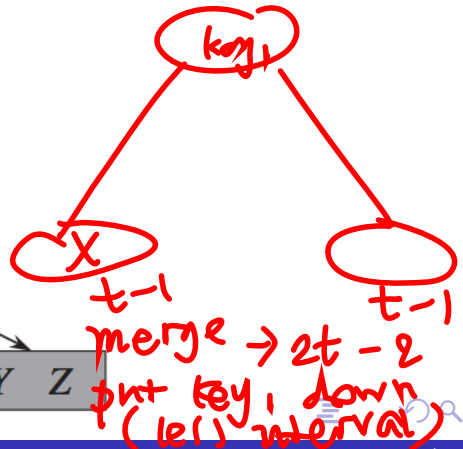
(b) *F* deleted: case 1





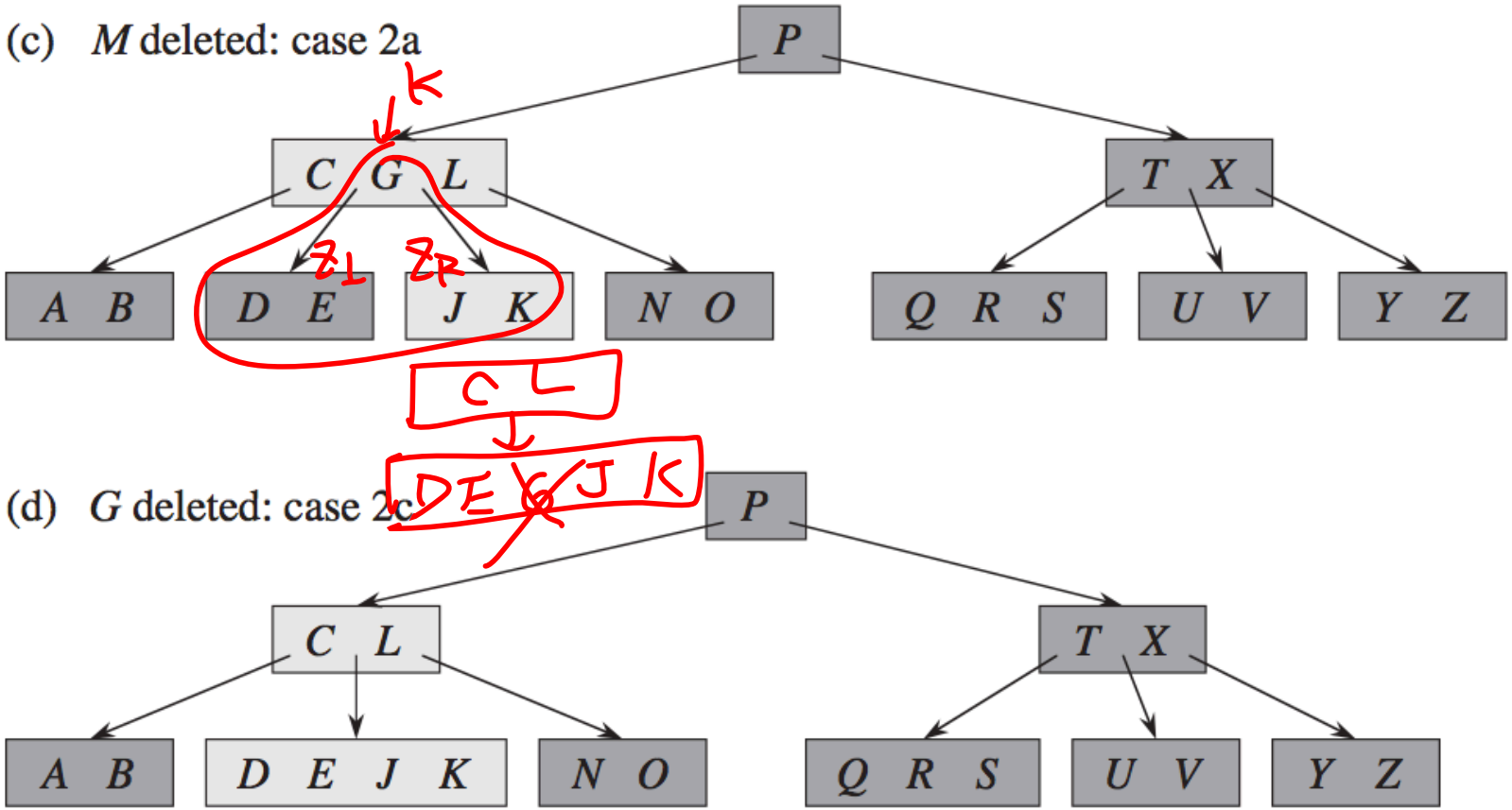
(c) *M* deleted: case 2a





B trees deletion

c. Otherwise, if both y and z have only $t - 1$ keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t - 1$ keys. Then free z and recursively delete k from y.

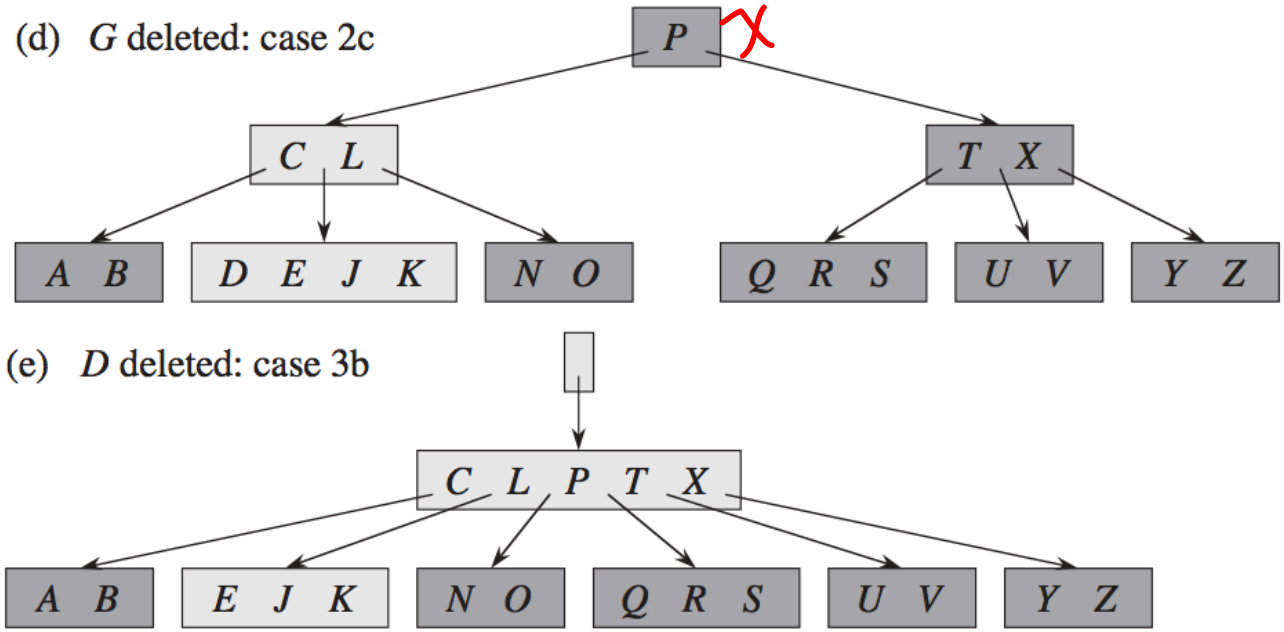


B trees deletion

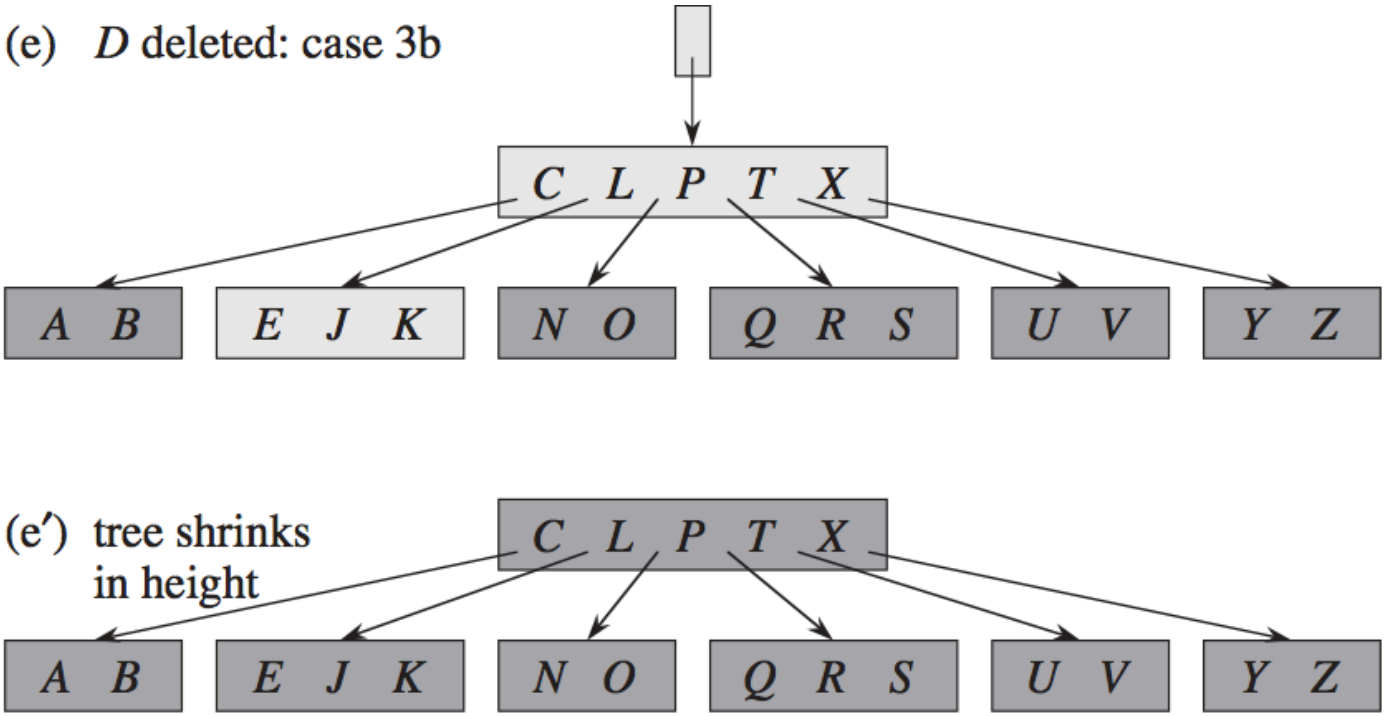
3. If the key k is not present in internal node x, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

borrow from siblings / merge

b. If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t-1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

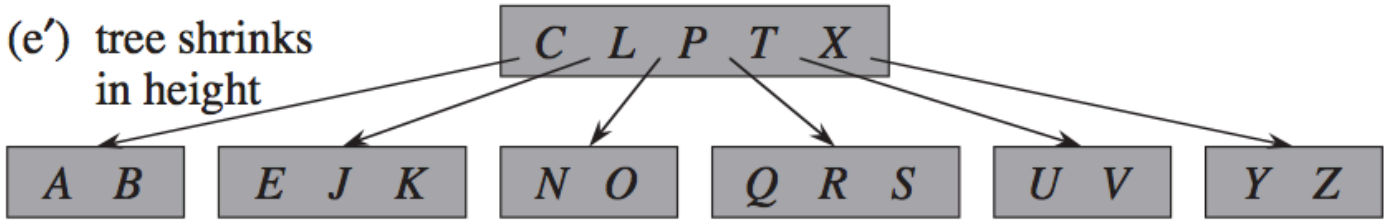


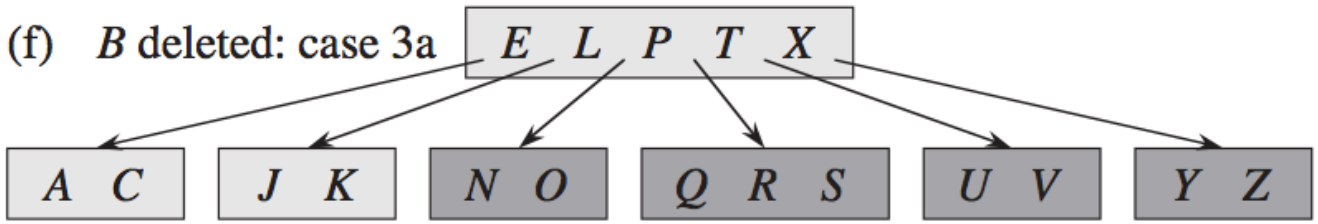
B trees deletion



B trees deletion

a. If $x.c_i$ has only $t - 1$ keys but has an immediate sibling with at least t keys, give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.



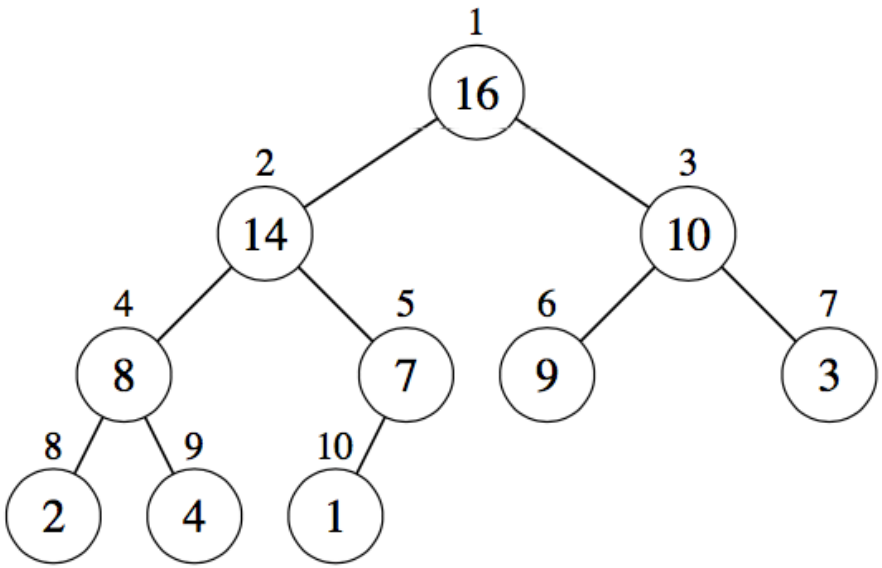


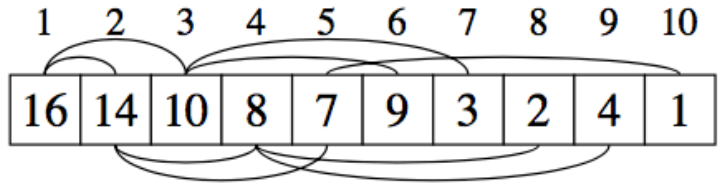
Running time: $O(th) = O(t \log_t n)$

Heap data structure

- Heap A (*not* garbage-collected storage) is a nearly complete binary tree.
  - ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.
  - ***Height*** of heap = height of root = $\Theta(\lg n)$.
- A heap can be stored as an array A.
  - Root of tree is $A[1]$.
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$.
  - Left child of $A[i] = A[2i]$.
  - Right child of $A[i] = A[2i + 1]$.
  - Computing is fast with binary representation implementation.

Example





Heaps property

- For max-heaps (largest element at root), ***max-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
- For min-heaps (smallest element at root), ***min-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.

By induction and transitivity of $\leq$, the max-heap property guarantees that the maximum element of a max-heap is at the root. Similar argument for min-heaps.

The heapsort algorithm we'll show uses max-heaps.

Note: In general, heaps can be k-ary tree instead of binary.

Maintaining heaps property

MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.

- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume left and right subtrees of i are max-heaps.
- After MAX-HEAPIFY, subtree rooted at i is a max-heap.

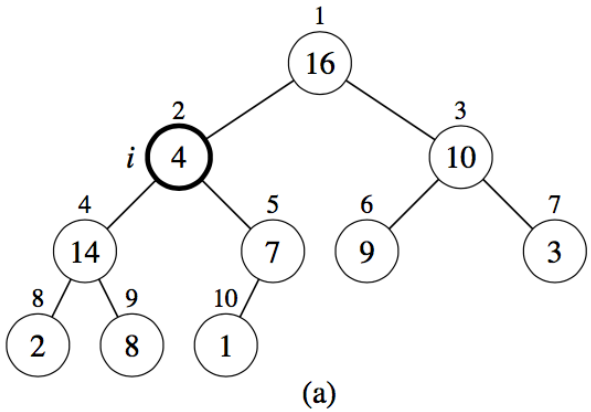
MAX-HEAPIFY(A, i, n)
$l \leftarrow$ LEFT(i)
$r \leftarrow$ RIGHT(i)
if $l \leq n$ and $A[l] > A[i]$
  then $largest \leftarrow l$
  else $largest \leftarrow i$
if $r \leq n$ and $A[r] > A[largest]$
  then $largest \leftarrow r$
if $largest \neq i$
  then exchange $A[i] \leftrightarrow A[largest]$
    MAX-HEAPIFY$(A, largest, n)$

How it works

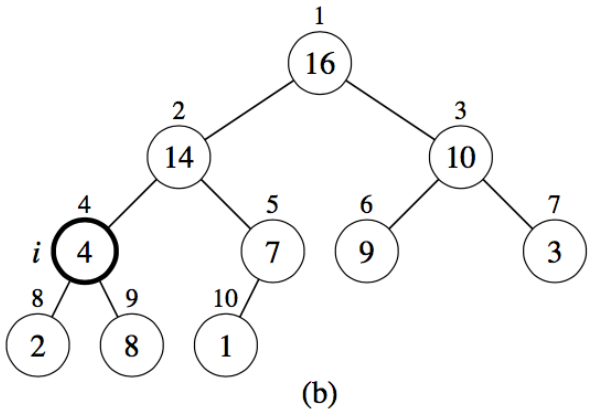
The way MAX-HEAPIFY works:

- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$.
- If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

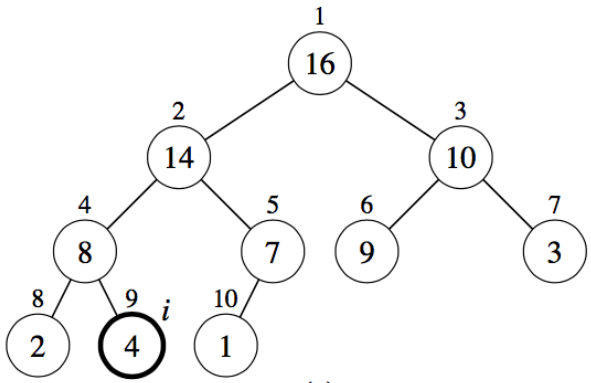
Example



(a)



(b)



(c)

- Node 2 violates the max-heap property.
- Compare node 2 with its children, and then swap it with the larger of the two children.
- Continue down the tree, swapping until the value is properly placed at the root of a subtree that is a max-heap. In this case, the max-heap is a leaf.

Time: $O(\lg n)$.

Builde a heap

BUILD-MAX-HEAP(A, n)

for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1

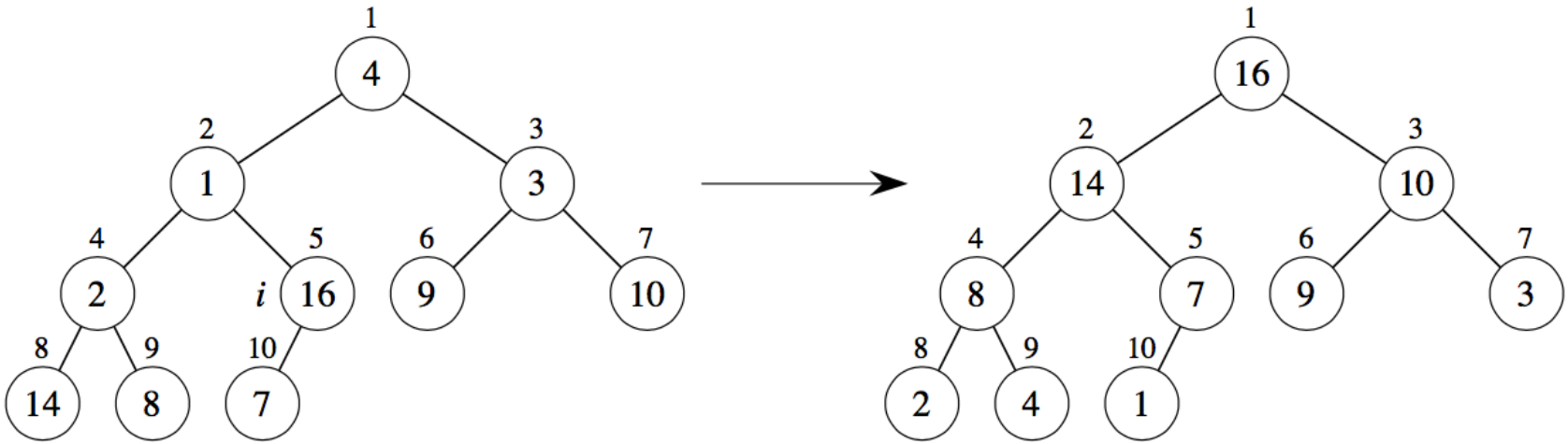
    do MAX-HEAPIFY(A, i, n)

Build a heap

Example: Building a max-heap from the following unsorted array results in the first heap example.

- i starts off as 5.
- MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7



Builde a heap

Correctness

Loop invariant: At start of every iteration of **for** loop, each node $i + 1$, $i + 2, \ldots, n$ is root of a max-heap.

Initialization: By Exercise 6.1-7, we know that each node $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, $\ldots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the **for** loop, the invariant is initially true.

Maintenance: Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i+1, i+2, \ldots, n$ are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root. Decrementing i reestablishes the loop invariant at each iteration.

Termination: When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

Time analysis for building a heap

- ***Simple bound:*** $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$. (Note: A good approach to analysis in general is to start by proving easy bound, then try to tighten it.)
- ***Tighter analysis:*** Observation: Time to run MAX-HEAPIFY is linear in the height of the node it's run on, and most nodes have small heights. Have $\leq \lceil n/2^{h+1} \rceil$ nodes of height h (see Exercise 6.3-3), and height of heap is $\lfloor \lg n \rfloor$ (Exercise 6.1-2).

  The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, so the total cost of BUILD-MAX-HEAP is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) .$$

Time analysis for building a heap

Evaluate the last summation by substituting $x = 1/2$ in the formula (A.8) ($\sum_{k=0}^{\infty} kx^k$), which yields

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2}$$
$$= 2 .$$

Thus, the running time of BUILD-MAX-HEAP is $O(n)$.

Building a min-heap from an unordered array can be done by calling MIN-HEAPIFY instead of MAX-HEAPIFY, also taking linear time.

Heapsort

Given an input array, the heapsort algorithm acts as follows:

- Builds a max-heap from the array.
- Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
- "Discard" this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
- Repeat this "discarding" process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

Heapsort

HEAPSORT(A, n)

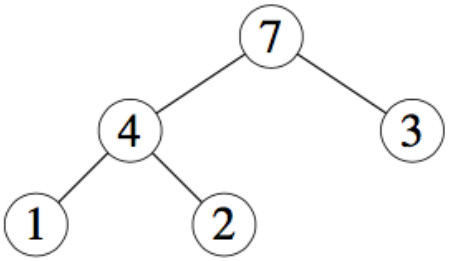
BUILD-MAX-HEAP(A, n)

for $i \leftarrow n$ **downto** 2

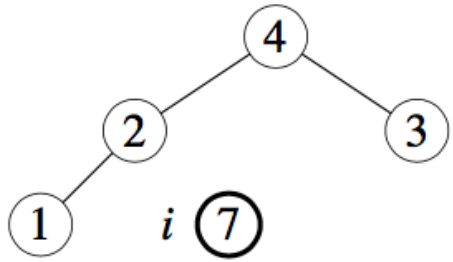
  do exchange $A[1] \leftrightarrow A[i]$

    MAX-HEAPIFY$(A, 1, i - 1)$

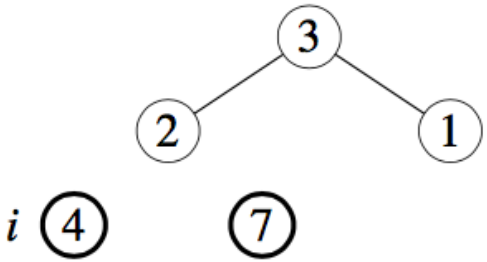
Heapsort example



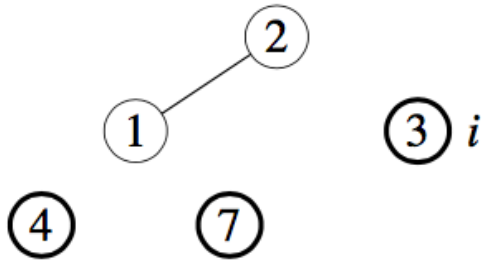
(a)



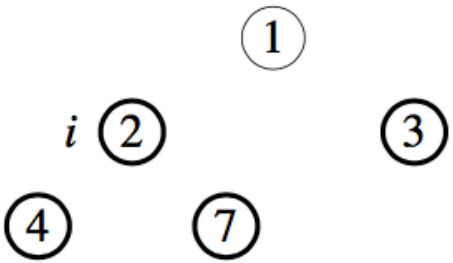
(b)



(c)



(d)



(e)

A	1	2	3	4	7

Heapsort analysis

Analysis

- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\lg n)$

Total time: $O(n \lg n)$.

Heap implementation of priority queue

- Maintains a dynamic set S of elements.
- Each set element has a ***key***—an associated value.
- Max-priority queue supports dynamic-set operations:
  - INSERT(S, x): inserts element x into set S.
  - MAXIMUM(S): returns element of S with largest key.
  - EXTRACT-MAX(S): removes and returns element of S with largest key.
  - INCREASE-KEY(S, x, k): increases value of element x's key to k. Assume $k \geq x$'s current key value.
- Example max-priority queue application: schedule jobs on shared computer.

Heap implementation of priority queue

- Min-priority queue supports similar operations:
  - INSERT(S, x): inserts element x into set S.
  - MINIMUM(S): returns element of S with smallest key.
  - EXTRACT-MIN(S): removes and returns element of S with smallest key.
  - DECREASE-KEY(S, x, k): decreases value of element x's key to k. Assume $k \leq x$'s current key value.
- Example min-priority queue application: event-driven simulator.

Find the max

Finding the maximum element

Getting the maximum element is easy: it's the root.

HEAP-MAXIMUM(A)
return $A[1]$

Time: $\Theta(1)$.

Extract the max

Given the array A:

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

HEAP-EXTRACT-MAX(A, n)
if $n < 1$
  then error "heap underflow"
$max \leftarrow A[1]$
$A[1] \leftarrow A[n]$
MAX-HEAPIFY$(A, 1, n-1)$ $\triangleright$ remakes heap
return max

Analysis of extracting max

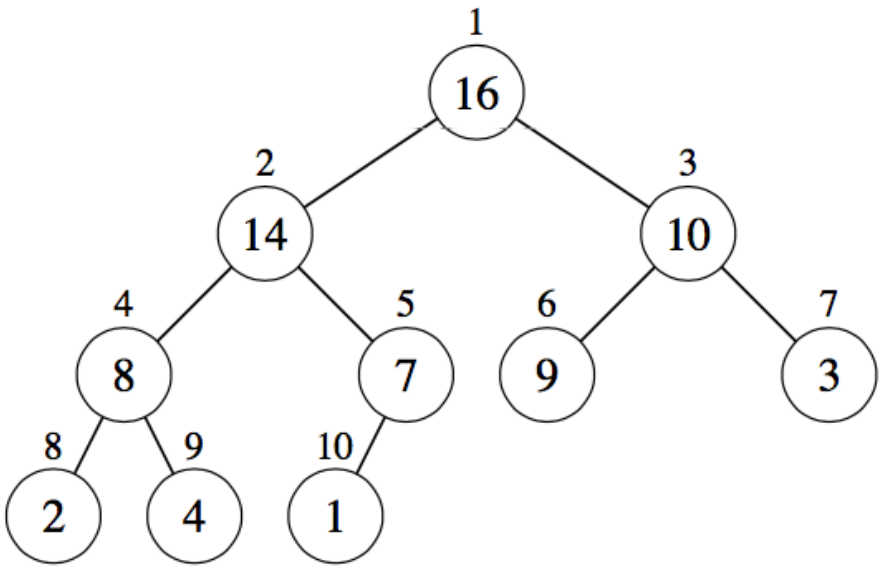
Analysis: constant time assignments plus time for MAX-HEAPIFY.

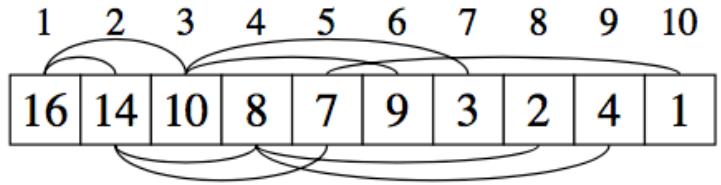
Time: $O(\lg n)$.

Example: Run HEAP-EXTRACT-MAX on first heap example.

- Take 16 out of node 1.
- Move 1 from node 10 to node 1.
- Erase node 10.
- MAX-HEAPIFY from the root to preserve max-heap property.
- Note that successive extractions will remove items in reverse sorted order.

Example





Insert key

Given set S, element x, and new key value k:

- Make sure $k \geq x$'s current key.
- Update x's key value to k.
- Traverse the tree upward comparing x to its parent and swapping keys if necessary, until x's key is smaller than its parent's key.

HEAP-INCREASE-KEY(A, i, key)

if $key < A[i]$
  then error "new key is smaller than current key"
$A[i] \leftarrow key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
  do exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
    $i \leftarrow \text{PARENT}(i)$

Heaps

Analysis: Upward path from node i has length $O(\lg n)$ in an n-element heap.

Time: $O(\lg n)$.

Example: Increase key of node 9 in first heap example to have value 15. Exchange keys of nodes 4 and 9, then of nodes 2 and 4.

Heaps

Given a key k to insert into the heap:

- Insert a new node in the very last position in the tree with key $-\infty$.
- Increase the $-\infty$ key to k using the HEAP-INCREASE-KEY procedure defined above.

MAX-HEAP-INSERT(A, key, n)

$A[n+1] \leftarrow -\infty$

HEAP-INCREASE-KEY$(A, n+1, key)$

Analysis: constant time assignments + time for HEAP-INCREASE-KEY.

Time: $O(\lg n)$.

Min-priority queue operations are implemented similarly with min-heaps.