# Data Structures

Gabriel Istrate

May 19, 2025

# How to augment a data structure

- Four steps:
- 1. Choose underlying data structure.
- 2. Determine additional information to be maintained.
- 3. Verify that additional information can be maintained in the D.S. operations.
- 4. develop new operations required by new fields.

# How to augment a data structure (II)

1. Choose red-black trees. Clue: supports other dynamic set operations on total order: MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR.
2. We didn't need field size to implement OS-SELECT, OS-RANK, but then operations wouldn't run in $O(\log n)$ time. Additional information to be maintained: sometimes pointer rather than data.
3. Ideally only a few elements need to be updated to maintain D.S. E.g. if we simply stored in each node it rank in the tree then OS-SELECT and OS-UPDATE would be efficient but inserting a smallest node causes changes in the whole tree.
4. Developed OS-SELECT, OS-RANK. Occasionally, instead of new operations, speed-up old ones.

# Augmenting red-black trees

Theorem

*Let $f$ be a field that augments a RB tree of $n$ nodes, and suppose the contents of $f$ for node $x$ can be computed in $O(1)$ using only information in node $x$, $left[x]$ and $right[x]$, including $f[left[x]]$ and $f[right[x]]$. Then we can maintain the values of $f$ in all nodes in $T$ during insertion and deletion without asymptotically affecting $O(\log n)$ performance.*

Proof idea: change in field $f$ at a node $x$ propagates only to ancestors of $x$ in the tree.

# Interval trees

- closed interval: $[t_1, t_2]$. Also open, half-open intervals.
- $i = [t_1, t_2]$. $low[i] = t_1$, $high[i] = t_2$.
- $i$ and $i'$ overlap if $i \bigcap i' \neq \emptyset$. That is $low[i] \leq high[i']$ and $low[i'] \leq high[i]$.
- Want: Data structure representing a dynamic set of intervals.
- Must support the following operations:
- $INTERVAL - INSERT(T, x)$: adds element $x$, whose $int$ field contains an interval.
- $INTERVAL - DELETE(T, x)$: removes element $x$ from $T$.
- $INTERVAL - SEARCH(T, i)$: return pointer to an element $x$ such that $int[x]$ overlaps $i$, or $nil$ if no such element found.

# Intervals

- Any two intervals satisfy interval trichotomy: three alternatives:
  1. $i$ and $i'$ overlap.
  2. $i$ is to the left of $i'$ ($high[i] < low[i']$).
  3. $i$ is to the right of $i'$.($low[i] > high[i']$).

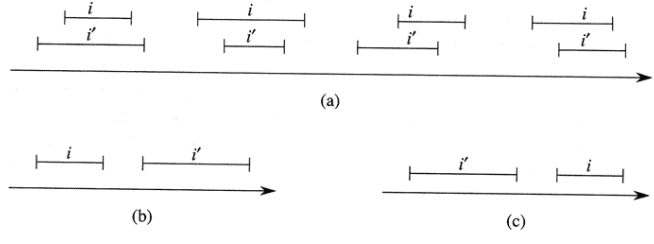



**Figure 14.3** The interval trichotomy for two closed intervals $i$ and $i'$. **(a)** If $i$ and $i'$ overlap, there are four situations; in each, $low[i] \le high[i']$ and $low[i'] \le high[i]$. **(b)** The intervals do not overlap, and $high[i] < low[i']$. **(c)** The intervals do not overlap, and $high[i'] < low[i]$.

# Interval trees: Implementation

1. Possible clue: intervals (partial) ordering. Might try to modify a total order. Then red-black tree. Each node $x$ stores an interval $int[x]$.
   - $key[x] = low[int[x]]$.
2. Additional info: $max[x]$, the maximum value of any endpoint of an interval stored in the subtree rooted at $x$.
3. Maintain info: $max[x] = \max(high[int[x]], max[left[x]], max[right[x]])$.
4. By applying previous theorem: insertion/deletion $O(\log n)$ while maintaining $max[x]$.

# Interval tree

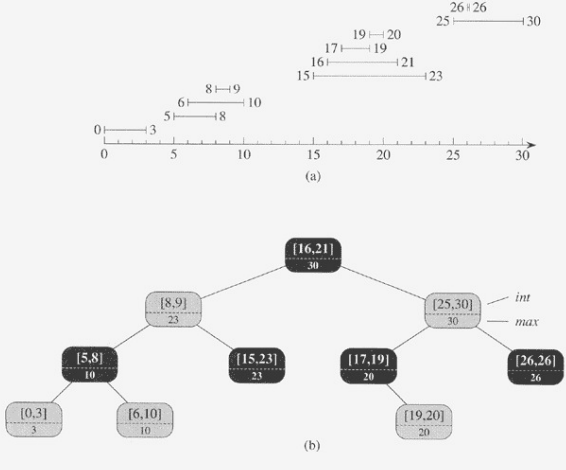



**Figure 14.4** An interval tree. **(a)** A set of 10 intervals, shown sorted bottom to top by left endpoint. **(b)** The interval tree that represents them. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.

# INTERVAL-SEARCH

- finds a node in tree $T$ whose interval overlaps interval $i$, returns sentinel node $nil[T]$ if no overlapping interval found.
- Search starts at the root and proceeds downwards.
- Chooses *left* or *right* subtree based on the maximum element in the left subtree of $x$.
- If $max[left[x]]$ is $\geq low[i]$ (of course, $left[x] \neq nil[T]$) go left.
- otherwise go right.
- takes $O(\log n)$ time since each basic loop takes $O(1)$ time and the height of the RB tree is $O(\log n)$.

# INTERVAL-SEARCH

```
INTERVAL-SEARCH(T, i)
1  x ← root[T]
2  while x ≠ nil[T] and i does not overlap int[x]
3       do if left[x] ≠ nil[T] and max[left[x]] ≥ low[i]
4             then x ← left[x]
5             else x ← right[x]
6  return x
```

# Correctness of INTERVAL-SEARCH

- Why is it enough to examine a single path ?
- Idea: search proceeds in a "safe direction".
- INVARIANT: If tree $T$ contains an interval that overlaps $i$ then there is such an interval in the subtree rooted at $x$.
- Initialization: clearly satisfied, $x = root[T]$.
- Either line 4 or line 5 executed.
- Line 5 executed: because $left[x] = nil[T]$ or $max[left[x]] < low[i]$. The subtree rooted at left[x] does not contain any interval that overlaps $i$.
- If such an interval is found in $T$, it must be in $right[x]$.

# Correctness of INTERVAL-SEARCH

- Line 4 executed: contrapositive of loop invariant holds.
- If there is no such an interval in the subtree rooted at $left[x]$ then there is no such interval in tree $T$.
- Since line 4 executed $max[left[x]] \geq low[i]$. There exists $i'$ with $high[i'] = max[left[x]] \geq low[i]$.
- $i$ and $i'$ do not overlap, by assumption. By trichotomy $high[i] < low[i']$.
- $i''$ interval in $right[x]$. Intervals keyed on the low endpoints.
- $high[i] < low[i'] \leq low[i'']$.
- Conclusion: no interval in $right[x]$ (and thus in $T$) overlaps $i$.

# Where are we:

Next

- Data Structures for external memory: B-Trees.

# Next: B-Trees

Outline:

- Search in secondary storage
- B-Trees
  - properties
  - search
  - insertion

# Complexity Model

# Complexity Model

- Basic assumption so far: *data structures fit completely in main memory (RAM)*
  - all basic operations have the same cost
    - even this is a rough approximation, since the main-memory system is not at all "flat"

# Complexity Model

- Basic assumption so far: *data structures fit completely in main memory (RAM)*
  - all basic operations have the same cost
    - even this is a rough approximation, since the main-memory system is not at all "flat"
- However, some applications require more storage than what fits in main memory
  - we must use data structures that reside in *secondary storage* (i.e., disk)

# Complexity Model

- Basic assumption so far: *data structures fit completely in main memory (RAM)*
  - all basic operations have the same cost
    - even this is a rough approximation, since the main-memory system is not at all "flat"
- However, some applications require more storage than what fits in main memory
  - we must use data structures that reside in *secondary storage* (i.e., disk)

***Disk is 10,000–100,000 times slower than RAM***

# Idea

# Idea

- In a balanced *binary* tree, $n$ keys require a tree of height $h = \lfloor \log_2 n \rfloor$
  - all the important operations require access to $O(h)$ nodes
  - each one accounting for *one or very few* basic operations

# Idea

- In a balanced *binary* tree, $n$ keys require a tree of height $h = \lfloor \log_2 n \rfloor$
  - all the important operations require access to $O(h)$ nodes
  - each one accounting for *one or very few* basic operations

- **Idea:** store several keys and pointers to children nodes in a single node

# Idea

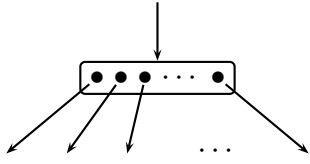
- In a balanced *binary* tree, $n$ keys require a tree of height $h = \lfloor \log_2 n \rfloor$
  - all the important operations require access to $O(h)$ nodes
  - each one accounting for *one or very few* basic operations

- **Idea:** store several keys and pointers to children nodes in a single node
  - in practice we ***increase the degree*** (or *branching factor*) of each node up to $d > 2$, so $h = \lfloor \log_d n \rfloor$
    - in practice $d$ can be as high as a few thousands

# Idea

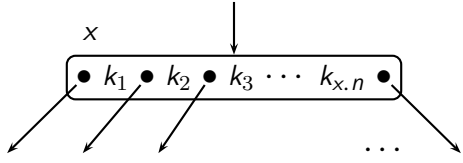
- In a balanced *binary* tree, $n$ keys require a tree of height $h = \lfloor \log_2 n \rfloor$
  - all the important operations require access to $O(h)$ nodes
  - each one accounting for *one or very few* basic operations

- **Idea:** store several keys and pointers to children nodes in a single node
  - in practice we ***increase the degree*** (or *branching factor*) of each node up to $d > 2$, so $h = \lfloor \log_d n \rfloor$
    - in practice $d$ can be as high as a few thousands

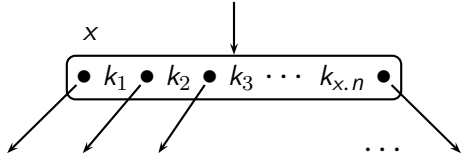


E.g., if $d = 1000$, then
***only three accesses*** $(h = 2)$
cover ***up to one billion keys***

# Definition of a B-Tree

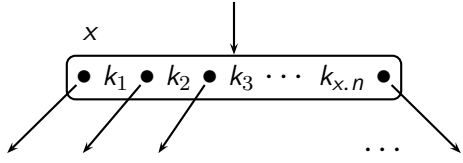


# Definition of a B-Tree



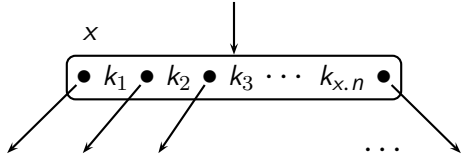

- Every node $x$ has the following fields
  - $x.n$ is the number of keys stored at each node

# Definition of a B-Tree



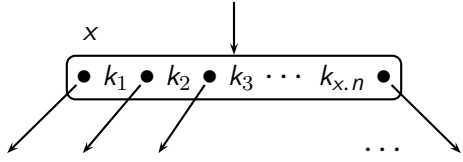

- Every node $x$ has the following fields
  - $x.n$ is the number of keys stored at each node
  - $x.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the $x.n$ keys stored in nondecreasing order

# Definition of a B-Tree



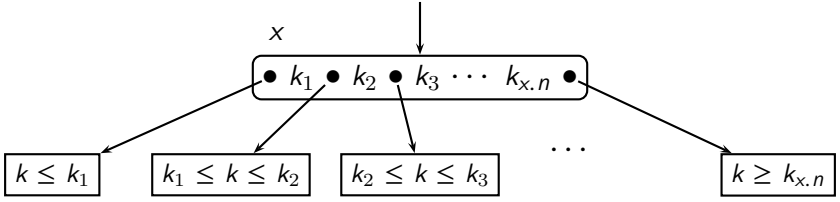

- Every node $x$ has the following fields
  - $x.n$ is the number of keys stored at each node
  - $x.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the $x.n$ keys stored in nondecreasing order
  - $x.leaf$ is a Boolean flag that is TRUE if $x$ is a *leaf node* or FALSE if $x$ is an *internal node*

# Definition of a B-Tree

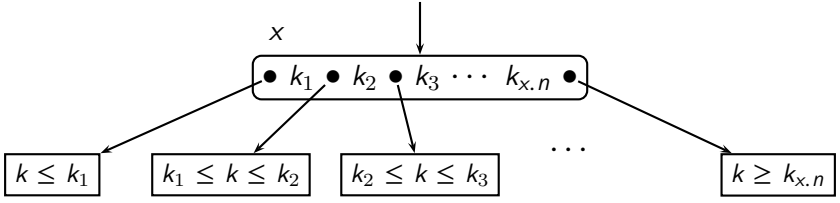



- Every node $x$ has the following fields
  - $x.n$ is the number of keys stored at each node
  - $x.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the $x.n$ keys stored in nondecreasing order
  - $x.leaf$ is a Boolean flag that is TRUE if $x$ is a *leaf node* or FALSE if $x$ is an *internal node*
  - $x.c[1], x.c[2], \ldots, x.c[x.n + 1]$ are the $x.n + 1$ pointers to its children, if $x$ is an *internal node*

# Definition of a B-Tree (2)

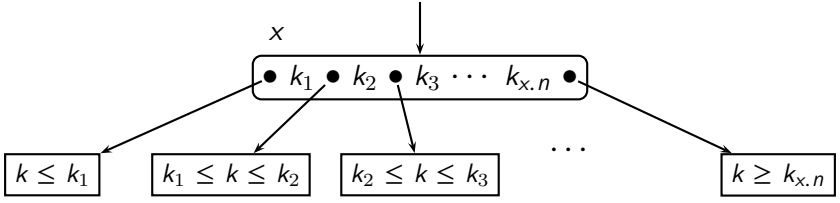


# Definition of a B-Tree (2)




- The keys $x.key[i]$ delimit the ranges of keys stored in each subtree

# Definition of a B-Tree (2)




- The keys $x.key[i]$ delimit the ranges of keys stored in each subtree

  $x.c[1] \longrightarrow$ subtree containing keys $k \leq x.key[1]$

  $x.c[2] \longrightarrow$ subtree containing keys $k$, $x.key[1] \leq k \leq x.key[2]$

  $x.c[3] \longrightarrow$ subtree containing keys $k$, $x.key[2] \leq k \leq x.key[3]$

  $\cdots$

  $x.c[x.n + 1] \longrightarrow$ subtree containing keys $k$, $k \geq x.key[x.n]$

# Definition of a B-Tree (3)

# Definition of a B-Tree (3)

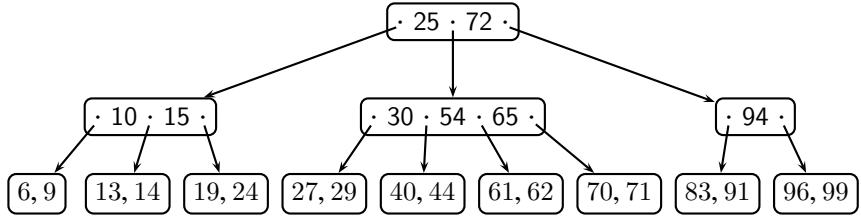
- ***All leaves have the same depth***

# Definition of a B-Tree (3)

- ***All leaves have the same depth***

- Let $t \geq 2$ be the ***minimum degree*** of the B-tree
  - every node other than the root must have ***at least*** $t - 1$ ***keys***
  - every node must contain ***at most*** $2t - 1$ ***keys***
    - a node is *full* when it contains exactly $2t - 1$ keys
    - a full node has $2t$ children

# Example


· 25 · 72 ·
· 10 · 15 ·
· 30 · 54 · 65 ·
· 94 ·
6, 9
13, 14
19, 24
27, 29
40, 44
61, 62
70, 71
83, 91
96, 99

# Search in B-Trees

# Search in B-Trees

```
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  if x.leaf
7        return NIL
8  else Disk-Read(x.c[i])
9        return B-Tree-Search(x.c[i], k)
```

# Height of a B-Tree

# Height of a B-Tree

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

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

# Height of a B-Tree

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

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

*Proof:*

  - $n \geq 1$, so the root has at least one key (and therefore two children)

# Height of a B-Tree

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

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

*Proof:*

- $n \geq 1$, so the root has at least one key (and therefore two children)
- every other node has at least $t$ children

# Height of a B-Tree

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

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

*Proof:*

- $n \geq 1$, so the root has at least one key (and therefore two children)
- every other node has at least $t$ children
- in the worst case, there are two subtrees (of the root) each one containing a total of $(n-1)/2$ keys, and each one consisting of $t$-degree nodes, with each node containing $t-1$ keys

# Height of a B-Tree

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

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

*Proof:*

- $n \geq 1$, so the root has at least one key (and therefore two children)
- every other node has at least $t$ children
- in the worst case, there are two subtrees (of the root) each one containing a total of $(n-1)/2$ keys, and each one consisting of $t$-degree nodes, with each node containing $t-1$ keys
- each subtree contains $1 + t + t^2 \cdots + t^{h-1}$ nodes, each one containing $t-1$ keys

# Height of a B-Tree

- **Theorem:** the height of a B-tree containing $n \geq 1$ keys and with a minimum degree $t \geq 2$ is

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

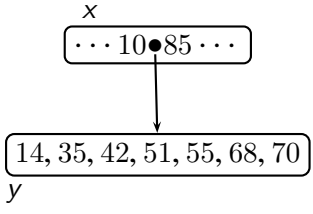
*Proof:*

- $n \geq 1$, so the root has at least one key (and therefore two children)
- every other node has at least $t$ children
- in the worst case, there are two subtrees (of the root) each one containing a total of $(n-1)/2$ keys, and each one consisting of $t$-degree nodes, with each node containing $t-1$ keys
- each subtree contains $1 + t + t^2 \cdots + t^{h-1}$ nodes, each one containing $t-1$ keys, so

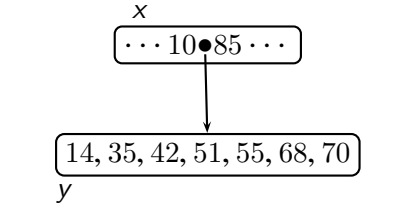
$$n \geq 1 + 2(t^h - 1)$$

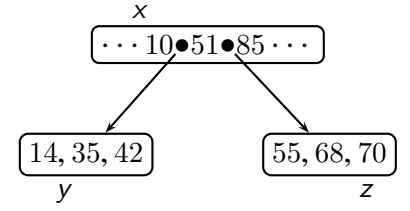
# Splitting

# Splitting

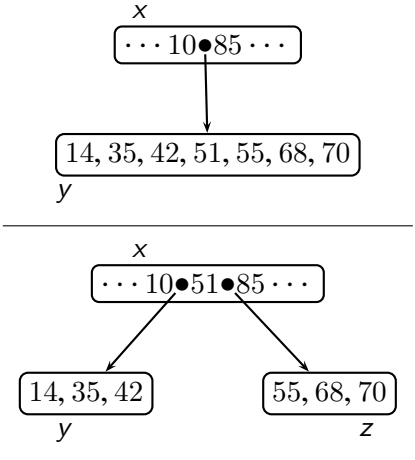


# Splitting



# Splitting




```
B-Tree-Split-Child(x, i, y)
 1  z = Allocate-Node()
 2  z.leaf = y.leaf
 3  z.n = t − 1
 4  for j = 1 to t − 1
 5        z.key[j] = y.key[j + t]
 6  if not y.leaf
 7        for j = 1 to t
 8              z.c[j] = y.c[j + t]
 9  y.n = t − 1
10  for j = x.n + 1 downto i + 1
11        x.c[j + 1] = x.c[j]
12  for j = x.n downto i
13        x.key[j + 1] = x.key[j]
14  x.key[i] = y.key[t]
15  x.n = x.n + 1
16  Disk-Write(y)
17  Disk-Write(z)
18  Disk-Write(x)
```

# Complexity of B-Tree-Split-Child

- What is the complexity of **B-Tree-Split-Child**?

# Complexity of B-Tree-Split-Child

- What is the complexity of **B-Tree-Split-Child**?

- $\Theta(t)$ basic CPU operations

# Complexity of B-Tree-Split-Child

- What is the complexity of **B-Tree-Split-Child**?
- $\Theta(t)$ basic CPU operations
- 3 **Disk-Write** operations

```
B-Tree-Split-Child(x, i, y)
 1  z = Allocate-Node()
 2  z.leaf = y.leaf
 3  z.n = t − 1
 4  for j = 1 to t − 1
 5       x.key[j] = x.key[j + t]
 6  if not x.leaf
 7       for j = 1 to t
 8            z.c[j] = y.c[j + t]
 9  y.n = t − 1
10  for j = x.n + 1 downto i + 1
11       x.c[j + 1] = x.c[j]
12  for j = x.n downto i
13       x.key[j + 1] = x.key[j]
14  x.key[i] = y.key[t]
15  x.n = x.n + 1
16  Disk-Write(y)
17  Disk-Write(z)
18  Disk-Write(x)
```

# Insertion Under Non-Full Node

# Insertion Under Non-Full Node

```
B-Tree-Insert-Nonfull(x, k)
 1  i = x.n                                   // assume x is not full
 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              // child x.c[i] is full
14            B-Tree-Split-Child(x, i, x.c[i])
15            if k > x.key[i]
16                 i = i + 1
17       B-Tree-Insert-Nonfull(x.c[i], k)
```

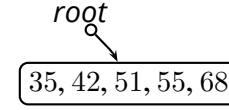
# Insertion Procedure

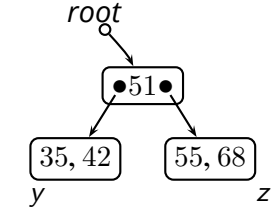
# Insertion Procedure

```
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, r)
 9        B-Tree-Insert-Nonfull(s, k)
10  else B-Tree-Insert-Nonfull(r, k)
```

# Insertion Procedure

```
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, r)
 9       B-Tree-Insert-Nonfull(s, k)
10  else B-Tree-Insert-Nonfull(r, k)
```



# Complexity of Insertion

- What is the complexity of **B-Tree-Insert**?

# Complexity of Insertion

- What is the complexity of **B-Tree-Insert**?
- $O(th) = O(t \log_t n)$ basic CPU steps operations

# Complexity of Insertion

- What is the complexity of **B-Tree-Insert**?
- $O(th) = O(t \log_t n)$ basic CPU steps operations
- $O(h) = O(\log_t n)$ disk-access operations

# Complexity of Insertion

- What is the complexity of **B-Tree-Insert**?
- $O(th) = O(t \log_t n)$ basic CPU steps operations
- $O(h) = O(\log_t n)$ disk-access operations
- The best value for $t$ can be determined according to
  - the ratio between CPU (RAM) speed and disk-access time
  - the *block-size* of the disk, which determines the maximum size of an object that can be accessed (read/write) in one shot

# Deletion

- Not hard, but more tedious.

# Deletion

- Not hard, but more tedious.
- Main problem: have to delete keys from every nodes, not just leaves

# Deletion

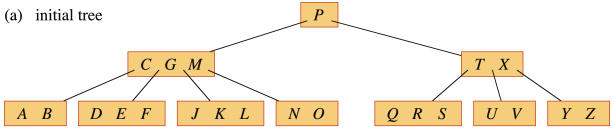
- Not hard, but more tedious.
- Main problem: have to delete keys from every nodes, not just leaves
- Insertion: merging nodes. Deletion: splitting nodes

# Deletion

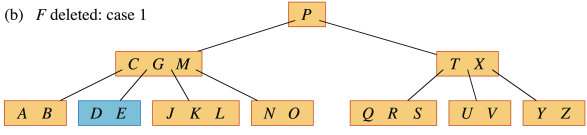
- Not hard, but more tedious.
- Main problem: have to delete keys from every nodes, not just leaves
- Insertion: merging nodes. Deletion: splitting nodes
- $O(h)$ disk operations. $O(th) = O(t \cdot log_t(n))$ CPU time.

# Deletion

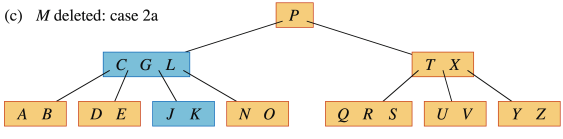
(a) initial tree



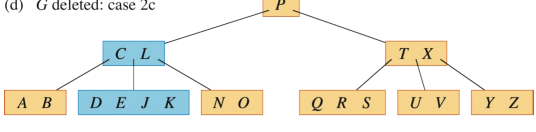

(b) *F* deleted: case 1




(c) *M* deleted: case 2a




(d) *G* deleted: case 2c



## Want to know more?