CS F364: Design & Analysis of Algorithms

Lecture-kt14: Amortized Analysis + B-Trees

Dr. Kamlesh Tiwari,
Assistant Professor,
Department of Computer Science and Information Systems,
BITS Pilani, Rajasthan-333031 INDIA

Feb 25, 2017 (Campus @ BITS-Pilani Jan-May 2017)

Binary Counter

- Consider incrementing a binary counter

Binary Counter

- Consider incrementing a binary counter

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Amortized Analysis

- In amortized analysis, time required to perform a sequence of data structure operation is averaged over all the operations performed

Amortized Analysis

- In amortized analysis, time required to perform a sequence of data structure operation is averaged over all the operations performed
- Sometime average cost of an operation is small, if we average over a sequence of operations

Amortized Analysis

- In amortized analysis, time required to perform a sequence of data structure operation is averaged over all the operations performed
- Sometime average cost of an operation is small, if we average over a sequence of operations
- Amortized analysis guarantees the average performance of each operation in the worst case

Increment-Binary-Counter

Algorithm 1: Increment-Binary-Counter(A)

```
1 i = 0
2 while i < length[A] and A[i] = 1 do
3     A[i] = 0
4     i = i + 1
5 if i < length[A] then
6     A[i] = 1
```

Increment-Binary-Counter

Algorithm 2: Increment-Binary-Counter(A)

```
1 i = 0
2 while i < length[A] and A[i] = 1 do
3     A[i] = 0
4     i = i + 1
5 if i < length[A] then
6     A[i] = 1
```

- A[1] flips each time the procedure to increment is called

Increment-Binary-Counter

Algorithm 3: Increment-Binary-Counter(A)

```
1 i = 0
2 while i < length[A] and A[i] = 1 do
3 |   A[i] = 0
4 |   i = i + 1
5 if i < length[A] then
6 |   A[i] = 1
```

- A[1] flips each time the procedure to increment is called
- A[2] flips every other time to A[1]

Increment-Binary-Counter

Algorithm 4: Increment-Binary-Counter(A)

```
1 i = 0
2 while i < length[A] and A[i] = 1 do
3 |   A[i] = 0
4 |   i = i + 1
5 if i < length[A] then
6 |   A[i] = 1
```

- A[1] flips each time the procedure to increment is called
- A[2] flips every other time to A[1]
- A[i] flips $\lfloor n/2^i \rfloor$ times in a sequence of n increments

Increment-Binary-Counter

Algorithm 5: Increment-Binary-Counter(A)

```
1 i = 0
2 while i < length[A] and A[i] = 1 do
3 |   A[i] = 0
4 |   i = i + 1
5 if i < length[A] then
6 |   A[i] = 1
```

- A[1] flips each time the procedure to increment is called
- A[2] flips every other time to A[1]
- A[i] flips $\lfloor n/2^i \rfloor$ times in a sequence of n increments
- Thus total number of flips

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Increment-Binary-Counter

Algorithm 6: Increment-Binary-Counter(A)

```
1 i = 0
2 while i < length[A] and A[i] = 1 do
3 |   A[i] = 0
4 |   i = i + 1
5 if i < length[A] then
6 |   A[i] = 1
```

- A[1] flips each time the procedure to increment is called
- A[2] flips every other time to A[1]
- A[i] flips $\lfloor n/2^i \rfloor$ times in a sequence of n increments
- Thus total number of flips

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

- Therefore, amortized cost of one operation is $O(n)/n = O(1)$

Stack with MULTIPOP operation

- PUSH(S,x): pushes object x onto stack S

Stack with MULTIPOP operation

- PUSH(S,x): pushes object x onto stack S
- POP(S): pops the top of stack S and returns the popped object

Stack with MULTIPOP operation

- PUSH(S,x): pushes object x onto stack S
- POP(S): pops the top of stack S and returns the popped object
- MULTIPOP(S,k):

Algorithm 9: MULTIPOP(S,k)

```
1 while NOT Stack-Empty(S) and k > 0 do
2     POP(S)
3     k = k- 1
```

- Running time of MULTIPOP(S,k) is $O(k)$

Stack with MULTIPOP operation

- PUSH(S,x): pushes object x onto stack S
- POP(S): pops the top of stack S and returns the popped object
- MULTIPOP(S,k):

Algorithm 10: MULTIPOP(S,k)

```
1 while NOT Stack-Empty(S) and k > 0 do
2     POP(S)
3     k = k- 1
```

- Running time of MULTIPOP(S,k) is $O(k)$
- Analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack

Stack with MULTIPOP operation

- PUSH(S,x): pushes object x onto stack S
- POP(S): pops the top of stack S and returns the popped object
- MULTIPOP(S,k):

Algorithm 11: MULTIPOP(S,k)

```
1 while NOT Stack-Empty(S) and k > 0 do
2     POP(S)
3     k = k- 1
```

- Running time of MULTIPOP(S,k) is $O(k)$
- Analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack
  - Worst-case cost of a MULTIPOP operation is $O(n)$

Stack with MULTIPOP operation

- PUSH(S,x): pushes object x onto stack S
- POP(S): pops the top of stack S and returns the popped object
- MULTIPOP(S,k):

Algorithm 12: MULTIPOP(S,k)

```
1 while NOT Stack-Empty(S) and k > 0 do
2     POP(S)
3     k = k- 1
```

- Running time of MULTIPOP(S,k) is $O(k)$
- Analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack
  - Worst-case cost of a MULTIPOP operation is $O(n)$
  - The worst-case time of any stack operation is therefore $O(n)$

Stack with MULTIPOP operation

- PUSH(S,x): pushes object x onto stack S
- POP(S): pops the top of stack S and returns the popped object
- MULTIPOP(S,k):

Algorithm 13: MULTIPOP(S,k)

```
1 while NOT Stack-Empty(S) and k > 0 do
2     POP(S)
3     k = k- 1
```

- Running time of MULTIPOP(S,k) is $O(k)$
- Analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack
  - Worst-case cost of a MULTIPOP operation is $O(n)$
  - The worst-case time of any stack operation is therefore $O(n)$
  - Hence a sequence of n operations costs $O(n^2)$

Stack with MULTIPOP operation

- PUSH(S,x): pushes object x onto stack S
- POP(S): pops the top of stack S and returns the popped object
- MULTIPOP(S,k):

Algorithm 14: MULTIPOP(S,k)

```
1 while NOT Stack-Empty(S) and k > 0 do
2     POP(S)
3     k = k- 1
```

- Running time of MULTIPOP(S,k) is $O(k)$
- Analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack
  - Worst-case cost of a MULTIPOP operation is $O(n)$
  - The worst-case time of any stack operation is therefore $O(n)$
  - Hence a sequence of n operations costs $O(n^2)$
  - This analysis is not tight

Stack with MULTIPOP operation

- Using amortized analysis, we can obtain a better upper bound

Stack with MULTIPOP operation

- Using amortized analysis, we can obtain a better upper bound
- Although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most $O(n)$

Stack with MULTIPOP operation

- Using amortized analysis, we can obtain a better upper bound
- Although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most $O(n)$
- Because, We can pop each object from the stack at most once for each time we have pushed it onto the stack

Stack with MULTIPOP operation

- Using amortized analysis, we can obtain a better upper bound
- Although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most $O(n)$
- Because, We can pop each object from the stack at most once for each time we have pushed it onto the stack
- Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n.

Stack with MULTIPOP operation

- Using amortized analysis, we can obtain a better upper bound
- Although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most $O(n)$
- Because, We can pop each object from the stack at most once for each time we have pushed it onto the stack
- Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n.
- For any value of n, any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time.

Stack with MULTIPOP operation

- Using amortized analysis, we can obtain a better upper bound
- Although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most $O(n)$
- Because, We can pop each object from the stack at most once for each time we have pushed it onto the stack
- Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n.
- For any value of n, any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time.
- The average cost of an operation is $O(n)/n = O(1)$

Stack with MULTIPOP operation

- Using amortized analysis, we can obtain a better upper bound
- Although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most $O(n)$
- Because, We can pop each object from the stack at most once for each time we have pushed it onto the stack
- Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n.
- For any value of n, any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time.
- The average cost of an operation is $O(n)/n = O(1)$

Note:

We did not used probabilistic reasoning. We showed a worst-case bound of $O(n)$ on a sequence of n operations. Dividing this total cost by n yielded the average cost per operation, or the amortized cost.

B-Tree

- B-trees are generalization of binary search trees

B-Tree

- B-trees are generalization of binary search trees
- They are balanced search trees designed to work well on disks or other direct-access secondary storage devices

B-Tree

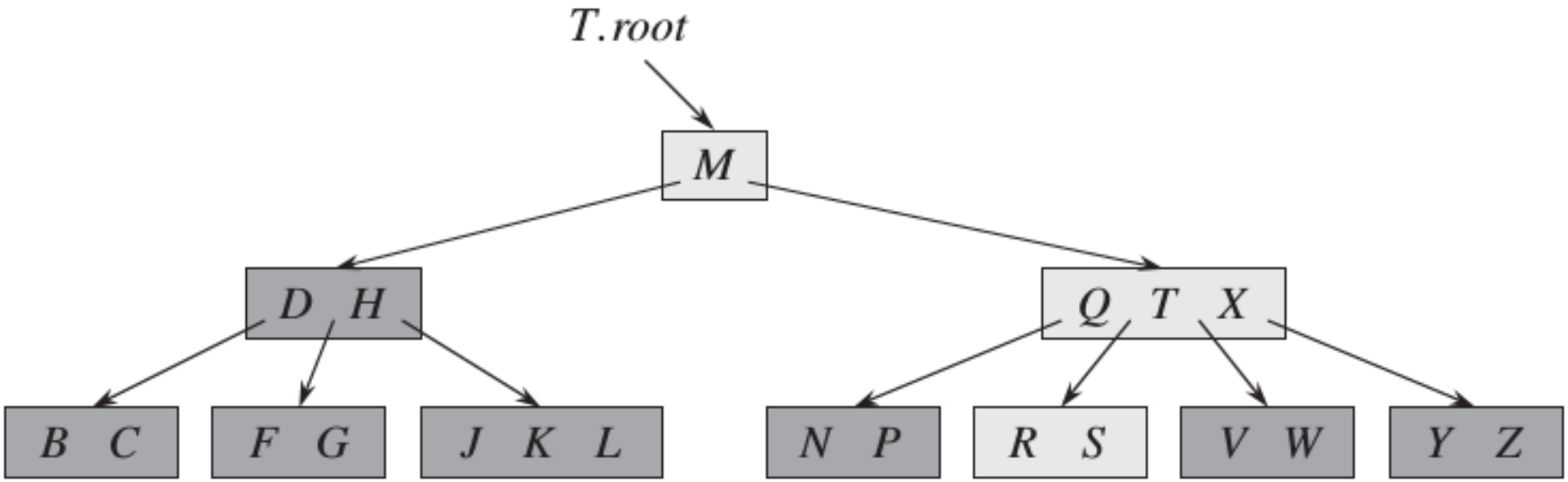
- B-trees are generalization of binary search trees
- They are balanced search trees designed to work well on disks or other direct-access secondary storage devices
- They are better at minimizing disk I/O operations

B-Tree

- B-trees are generalization of binary search trees
- They are balanced search trees designed to work well on disks or other direct-access secondary storage devices
- They are better at minimizing disk I/O operations
- Every n-node B-tree has height $O(\log n)$

B-Tree

- B-trees are generalization of binary search trees
- They are balanced search trees designed to work well on disks or other direct-access secondary storage devices
- They are better at minimizing disk I/O operations
- Every n-node B-tree has height $O(\log n)$



B-Tree

B-tree T is a rooted tree having following properties

B-Tree

B-tree T is a rooted tree having following properties

1. Every node x has the following attributes:

B-Tree

B-tree T is a rooted tree having following properties

1. Every node x has the following attributes:
    1. $x.n$ number of keys currently stored in x

B-Tree

B-tree T is a rooted tree having following properties

1. Every node x has the following attributes:
   1. $x.n$ number of keys currently stored in x
   2. $x.n$ keys are stored in non decreasing order $x.key_1 \leq x.key_2 \leq ... \leq x.key_{x.n}$

B-Tree

B-tree T is a rooted tree having following properties

1. Every node x has the following attributes:
   1. $x.n$ number of keys currently stored in x
   2. $x.n$ keys are stored in non decreasing order $x.key_1 \leq x.key_2 \leq ... \leq x.key_{x.n}$
2. Each internal node x also contains $x.n + 1$ pointers, $x.c_1, x.c_2, ..., x.c_{x.n+1}$

B-Tree

B-tree T is a rooted tree having following properties

1. Every node x has the following attributes:
   1. $x.n$ number of keys currently stored in x
   2. $x.n$ keys are stored in non decreasing order $x.key_1 \leq x.key_2 \leq ... \leq x.key_{x.n}$
2. Each internal node x also contains $x.n + 1$ pointers, $x.c_1, x.c_2, ..., x.c_{x.n+1}$
3. The key $x.key_i$ separate the range of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then $k_1 \leq x.key_1 \leq k_2 \leq x.key_2 ... \leq x.key_{x.n} \leq k_{x.n+1}$

B-Tree

B-tree T is a rooted tree having following properties

1. Every node x has the following attributes:
   1. $x.n$ number of keys currently stored in x
   2. $x.n$ keys are stored in non decreasing order $x.key_1 \leq x.key_2 \leq ... \leq x.key_{x.n}$
2. Each internal node x also contains $x.n + 1$ pointers, $x.c_1, x.c_2, ..., x.c_{x.n+1}$
3. The key $x.key_i$ separate the range of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then $k_1 \leq x.key_1 \leq k_2 \leq x.key_2 ... \leq x.key_{x.n} \leq k_{x.n+1}$
4. All leaves have same depth, which is tree's height

B-Tree

B-tree T is a rooted tree having following properties

1. Every node x has the following attributes:
   1. $x.n$ number of keys currently stored in x
   2. $x.n$ keys are stored in non decreasing order $x.key_1 \leq x.key_2 \leq ... \leq x.key_{x.n}$
2. Each internal node x also contains $x.n + 1$ pointers, $x.c_1, x.c_2, ..., x.c_{x.n+1}$
3. The key $x.key_i$ separate the range of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then $k_1 \leq x.key_1 \leq k_2 \leq x.key_2 ... \leq x.key_{x.n} \leq k_{x.n+1}$
4. All leaves have same depth, which is tree's height
5. $t \geq 2$ is **minimum degree** of B-tree

B-Tree

B-tree T is a rooted tree having following properties

1. Every node x has the following attributes:
   1. $x.n$ number of keys currently stored in x
   2. $x.n$ keys are stored in non decreasing order $x.key_1 \leq x.key_2 \leq ... \leq x.key_{x.n}$
2. Each internal node x also contains $x.n + 1$ pointers, $x.c_1, x.c_2, ..., x.c_{x.n+1}$
3. The key $x.key_i$ separate the range of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then $k_1 \leq x.key_1 \leq k_2 \leq x.key_2 ... \leq x.key_{x.n} \leq k_{x.n+1}$
4. All leaves have same depth, which is tree's height
5. $t \geq 2$ is **minimum degree** of B-tree
   1. Every node other then root must have $t - 1$ keys

B-Tree

B-tree T is a rooted tree having following properties

1. Every node x has the following attributes:
   1. $x.n$ number of keys currently stored in x
   2. $x.n$ keys are stored in non decreasing order $x.key_1 \leq x.key_2 \leq ... \leq x.key_{x.n}$
2. Each internal node x also contains $x.n + 1$ pointers, $x.c_1, x.c_2, ..., x.c_{x.n+1}$
3. The key $x.key_i$ separate the range of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then $k_1 \leq x.key_1 \leq k_2 \leq x.key_2 ... \leq x.key_{x.n} \leq k_{x.n+1}$
4. All leaves have same depth, which is tree's height
5. $t \geq 2$ is **minimum degree** of B-tree
   1. Every node other then root must have $t - 1$ keys
   2. Node can have at most $2t - 1$ keys (then it is **full**)

Height of B-Tree

B-Tree with n-keys has height $h \leq \log_t \frac{n+1}{2}$

Height of B-Tree

B-Tree with n-keys has height $h \leq \log_t \frac{n+1}{2}$

- Root can have minimum 1 key

Height of B-Tree

B-Tree with n-keys has height $h \leq \log_t \frac{n+1}{2}$

- Root can have minimum 1 key
- At depth 1, there would be minimum 2 nodes

Height of B-Tree

B-Tree with n-keys has height $h \leq \log_t \frac{n+1}{2}$

- Root can have minimum 1 key
- At depth 1, there would be minimum 2 nodes
- At depth 2, there would be minimum $2t$ nodes

Height of B-Tree

B-Tree with n-keys has height $h \leq \log_t \frac{n+1}{2}$

- Root can have minimum 1 key
- At depth 1, there would be minimum 2 nodes
- At depth 2, there would be minimum $2t$ nodes
- At depth 3, there would be minimum $2t^2$ nodes

Height of B-Tree

B-Tree with n-keys has height $h \leq \log_t \frac{n+1}{2}$

- Root can have minimum 1 key
- At depth 1, there would be minimum 2 nodes
- At depth 2, there would be minimum $2t$ nodes
- At depth 3, there would be minimum $2t^2$ nodes
- At depth h, there would be minimum $2t^{h-1}$ nodes

Height of B-Tree

B-Tree with n-keys has height $h \leq \log_t \frac{n+1}{2}$

- Root can have minimum 1 key
- At depth 1, there would be minimum 2 nodes
- At depth 2, there would be minimum $2t$ nodes
- At depth 3, there would be minimum $2t^2$ nodes
- At depth h, there would be minimum $2t^{h-1}$ nodes

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

Height of B-Tree

B-Tree with n-keys has height $h \leq \log_t \frac{n+1}{2}$

- Root can have minimum 1 key
- At depth 1, there would be minimum 2 nodes
- At depth 2, there would be minimum $2t$ nodes
- At depth 3, there would be minimum $2t^2$ nodes
- At depth h, there would be minimum $2t^{h-1}$ nodes

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$$1 + 2(t-1)\frac{t^h - 1}{t-1}$$

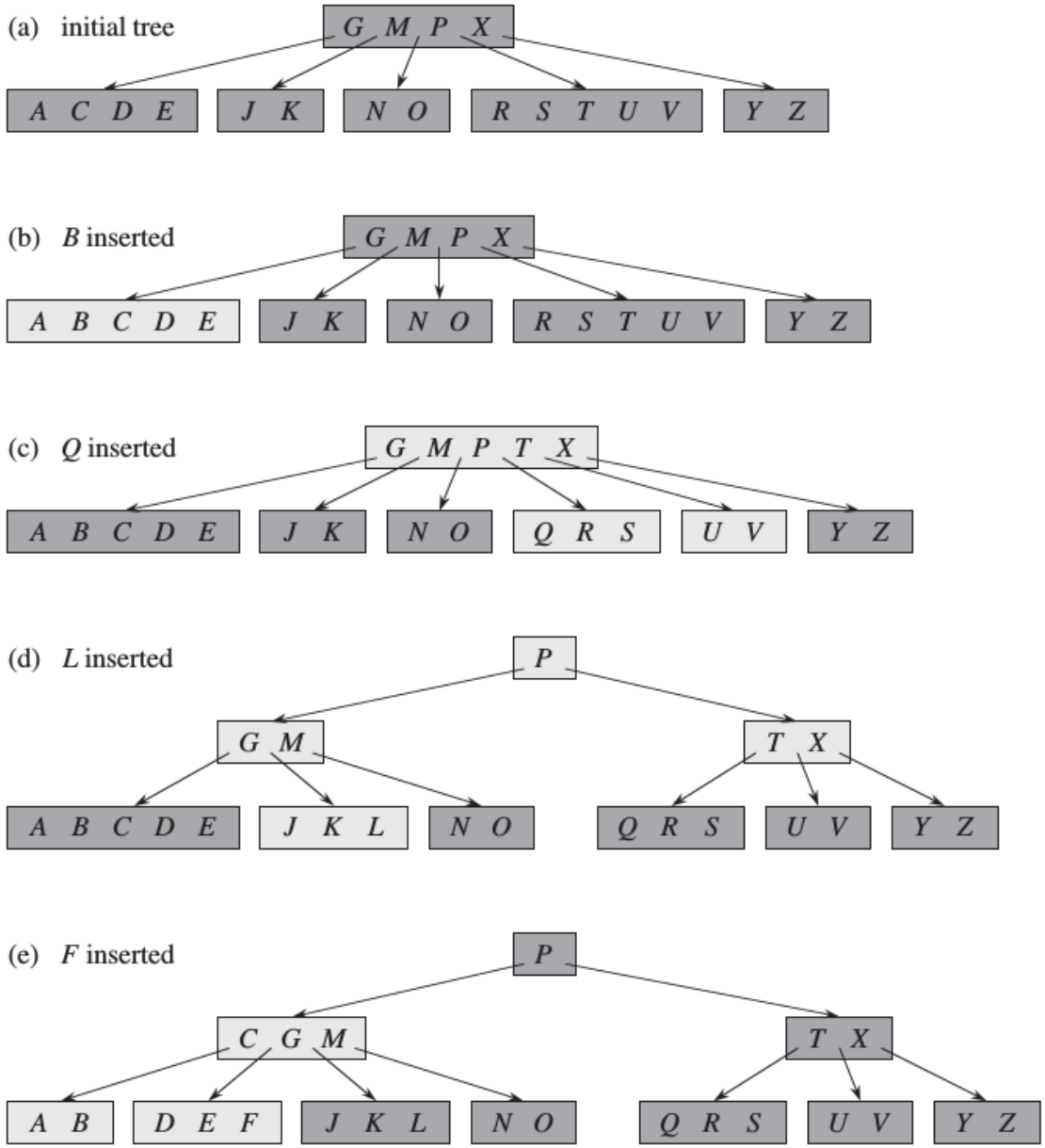
Height of B-Tree

B-Tree with n-keys has height $h \leq \log_t \frac{n+1}{2}$

- Root can have minimum 1 key
- At depth 1, there would be minimum 2 nodes
- At depth 2, there would be minimum $2t$ nodes
- At depth 3, there would be minimum $2t^2$ nodes
- At depth h, there would be minimum $2t^{h-1}$ nodes

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

Insertion of B-Tree



Thank You!

Thank you very much for your attention!

Queries ?