

Offline Caching Problem

Input: k : the size of cache

n : number of pages

$\rho_1, \rho_2, \rho_3, \dots, \rho_T \in [n]$: sequence of requests

Output: $i_1, i_2, i_3, \dots, i_t \in \{\text{hit}, \text{empty}\} \cup [n]$

- empty stands for an empty page
- “hit” means evicting no pages

Offline Caching Problem

Input: k : the size of cache

n : number of pages

$\rho_1, \rho_2, \rho_3, \dots, \rho_T \in [n]$: sequence of requests

$p_1, p_2, \dots, p_k \in \{\text{empty}\} \cup [n]$: initial set of pages in cache

Output: $i_1, i_2, i_3, \dots, i_t \in \{\text{hit}, \text{empty}\} \cup [n]$

- empty stands for an empty page
- “hit” means evicting no pages

Analysis of Greedy Algorithm

- **Safety:** Prove that the reasonable strategy is “safe” (key)
- **Self-reduce:** Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Analysis of Greedy Algorithm

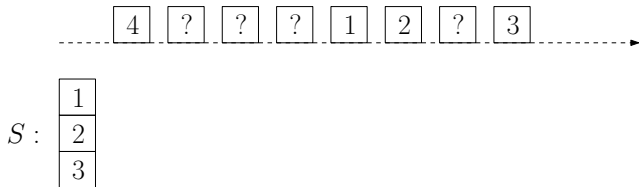
- **Safety:** Prove that the reasonable strategy is “safe” (key)
- **Self-reduce:** Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Lemma Assume at time 1 a page fault happens and there are no empty pages in the cache. Let p^* be the page in cache that is not requested until furthest in the future. **It is safe to evict p^* at time 1.**

Analysis of Greedy Algorithm

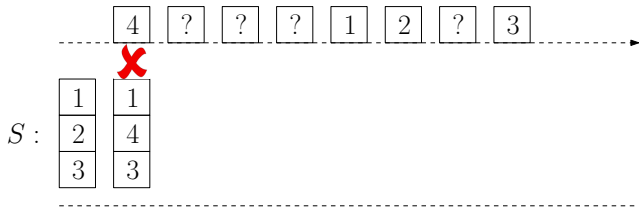
- **Safety:** Prove that the reasonable strategy is “safe” (key)
- **Self-reduce:** Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Lemma Assume at time 1 a page fault happens and there are no empty pages in the cache. Let p^* be the page in cache that is not requested until furthest in the future. **There is an optimum solution in which p^* is evicted at time 1.**



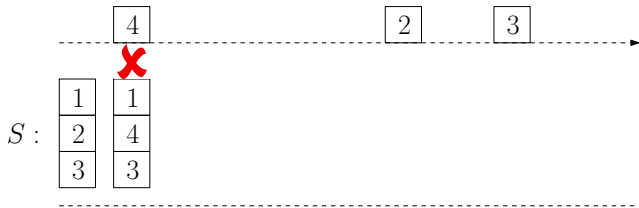
Proof.

- 1 S : any optimum solution
- 2 p^* : page in cache not requested until furthest in the future.
 - In the example, $p^* = 3$.



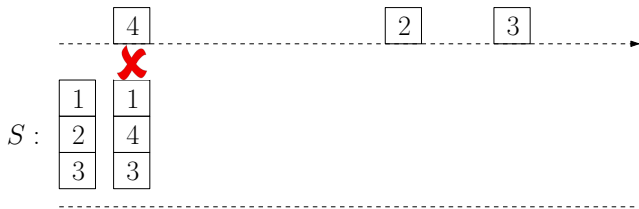
Proof.

- 1 S : any optimum solution
- 2 p^* : page in cache not requested until furthest in the future.
 - In the example, $p^* = 3$.
- 3 Assume S evicts some $p' \neq p^*$ at time 1; otherwise done.
 - In the example, $p' = 2$.

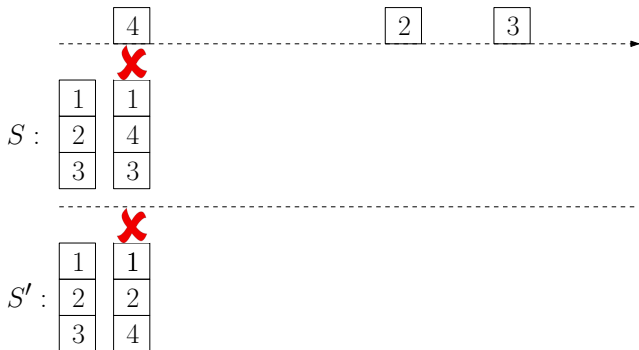


Proof.

- 1 S : any optimum solution
- 2 p^* : page in cache not requested until furthest in the future.
 - In the example, $p^* = 3$.
- 3 Assume S evicts some $p' \neq p^*$ at time 1; otherwise done.
 - In the example, $p' = 2$.

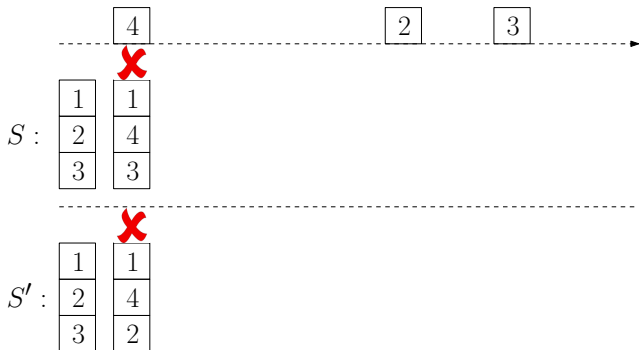


Proof.



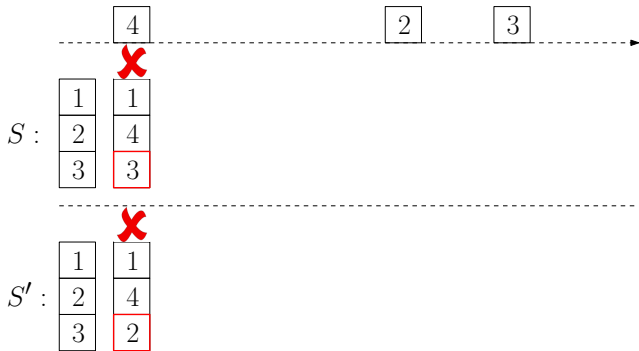
Proof.

- 1 Create S' . S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.



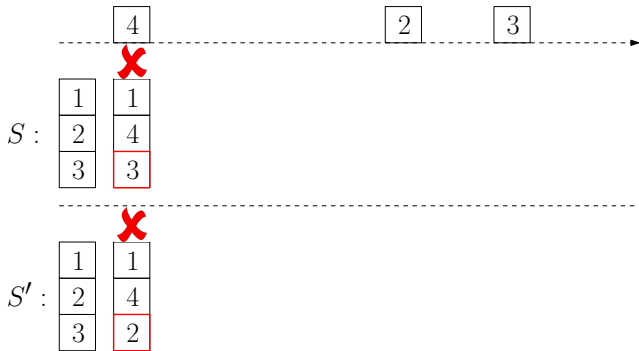
Proof.

- ④ Create S' . S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.



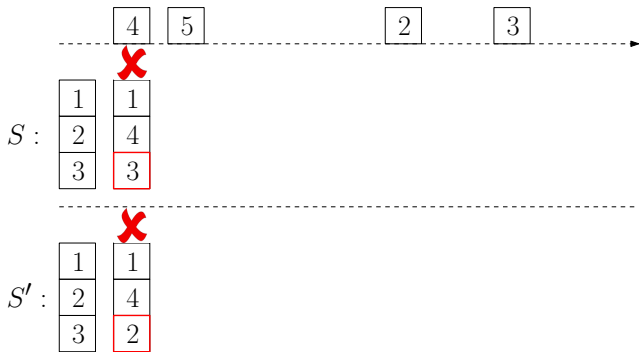
Proof.

- 4 Create S' . S' evicts $p^*(=3)$ instead of $p' (=2)$ at time 1.
- 5 After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p' (=2)$ and S contains $p^*(=3)$.



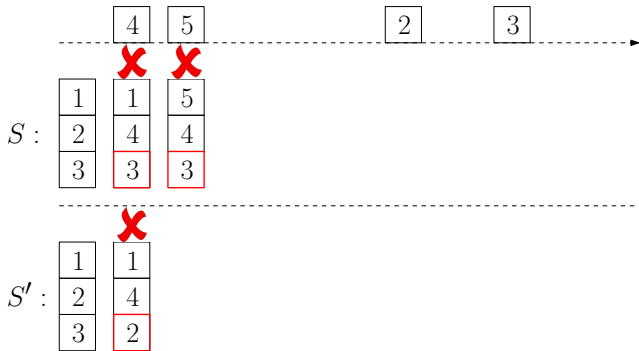
Proof.

- 4 Create S' . S' evicts $p^*(=3)$ instead of $p' (=2)$ at time 1.
- 5 After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p' (=2)$ and S contains $p^* (=3)$.
- 6 From now on, S' will “copy” S .



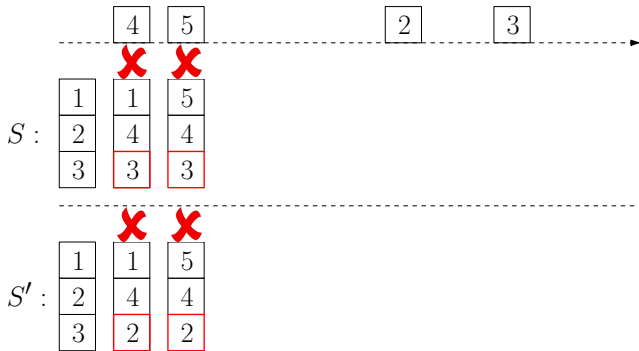
Proof.

- 4 Create S' . S' evicts $p^*(=3)$ instead of $p' (=2)$ at time 1.
- 5 After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p' (=2)$ and S contains $p^* (=3)$.
- 6 From now on, S' will “copy” S .



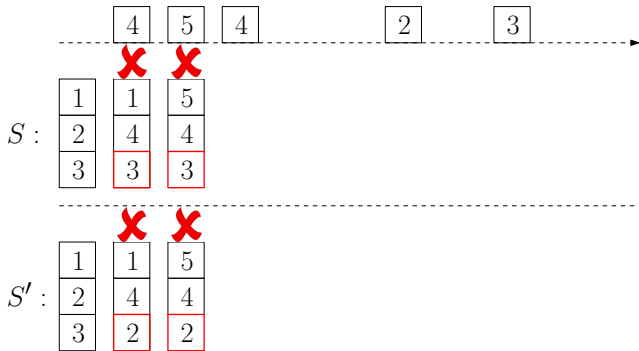
Proof.

- 4 Create S' . S' evicts $p^*(=3)$ instead of $p' (=2)$ at time 1.
- 5 After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p' (=2)$ and S contains $p^* (=3)$.
- 6 From now on, S' will “copy” S .



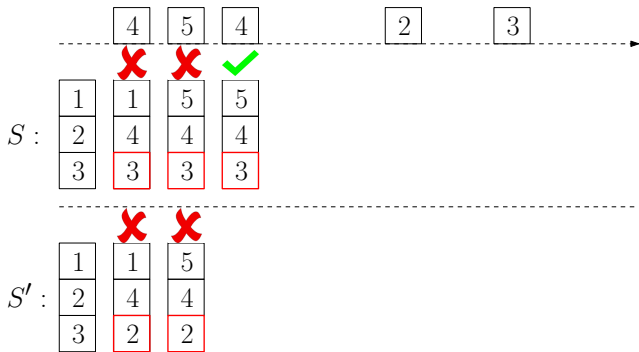
Proof.

- 4 Create S' . S' evicts $p^*(=3)$ instead of $p' (=2)$ at time 1.
- 5 After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p' (=2)$ and S contains $p^* (=3)$.
- 6 From now on, S' will “copy” S .



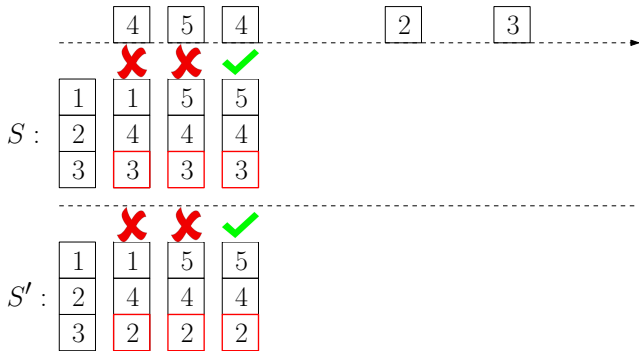
Proof.

- 4 Create S' . S' evicts $p^*(=3)$ instead of $p' (=2)$ at time 1.
- 5 After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p' (=2)$ and S contains $p^* (=3)$.
- 6 From now on, S' will “copy” S .



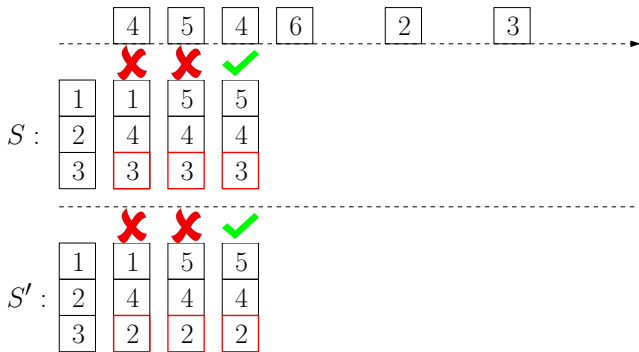
Proof.

- 4 Create S' . S' evicts $p^*(=3)$ instead of $p' (=2)$ at time 1.
- 5 After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p' (=2)$ and S contains $p^* (=3)$.
- 6 From now on, S' will “copy” S .



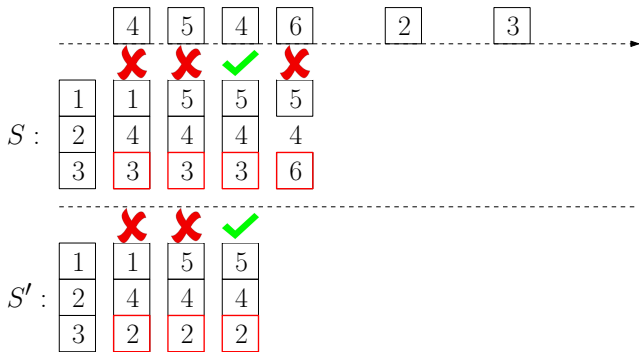
Proof.

- 4 Create S' . S' evicts $p^*(=3)$ instead of $p' (=2)$ at time 1.
- 5 After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p' (=2)$ and S contains $p^* (=3)$.
- 6 From now on, S' will “copy” S .



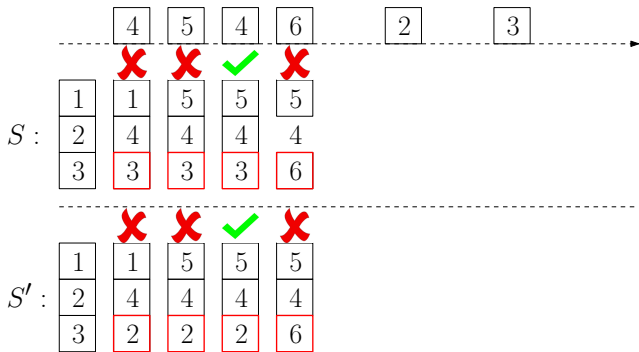
Proof.

- 4 Create S' . S' evicts $p^*(=3)$ instead of $p' (=2)$ at time 1.
- 5 After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p' (=2)$ and S contains $p^* (=3)$.
- 6 From now on, S' will “copy” S .



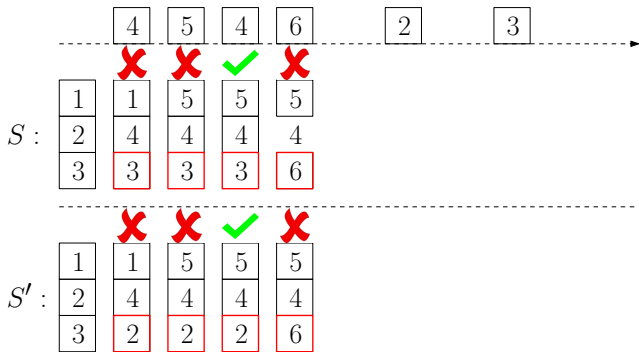
Proof.

- 4 Create S' . S' evicts $p^*(=3)$ instead of $p' (=2)$ at time 1.
- 5 After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p' (=2)$ and S contains $p^* (=3)$.
- 6 From now on, S' will “copy” S .

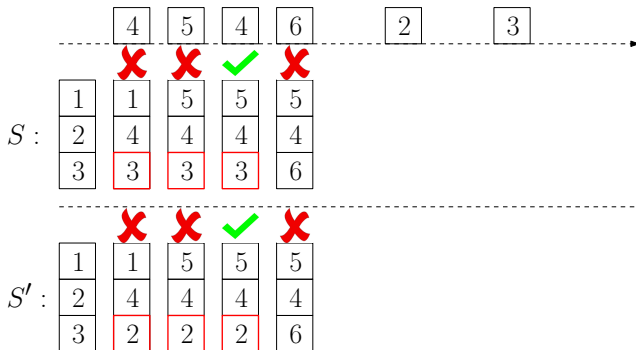


Proof.

- 4 Create S' . S' evicts $p^*(=3)$ instead of $p' (=2)$ at time 1.
- 5 After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p' (=2)$ and S contains $p^* (=3)$.
- 6 From now on, S' will “copy” S .

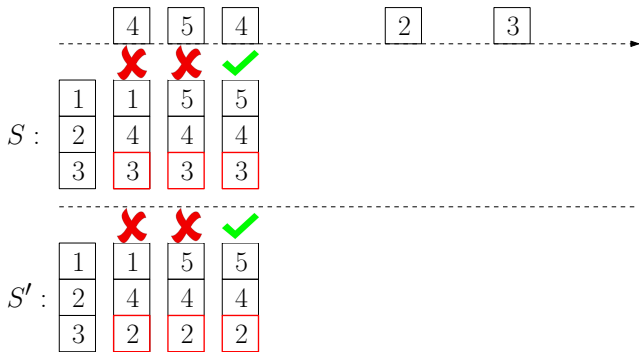


Proof.



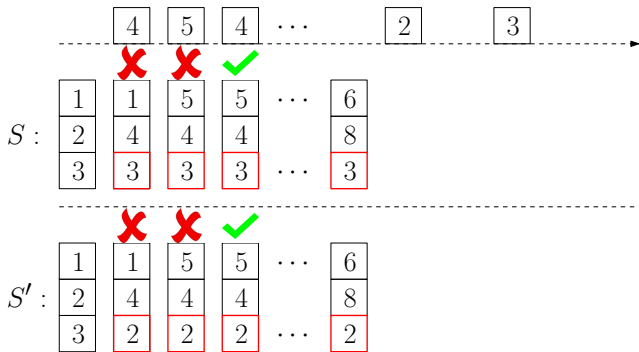
Proof.

- 7 If S evicted the page p^* , S' will evict the page p' . Then, the cache status of S and that of S' will be the same. S and S' will be exactly the same from now on.



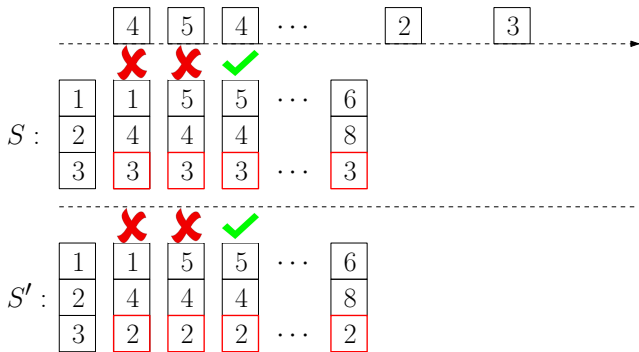
Proof.

- 7 If S evicted the page p^* , S' will evict the page p' . Then, the cache status of S and that of S' will be the same. S and S' will be exactly the same from now on.
- 8 Assume S did not evict $p^*(=3)$ before we see $p' (=2)$.

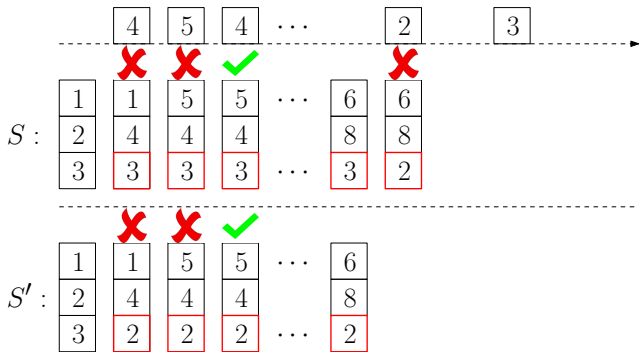


Proof.

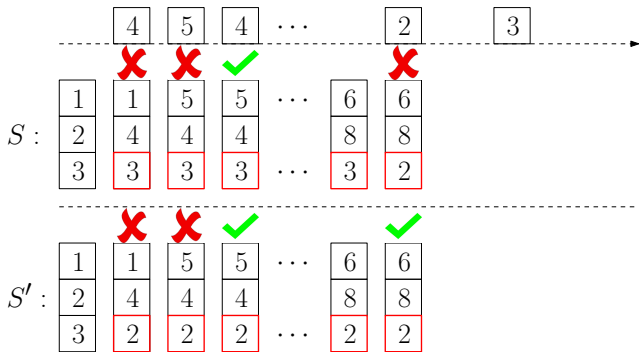
- 7 If S evicted the page p^* , S' will evict the page p' . Then, the cache status of S and that of S' will be the same. S and S' will be exactly the same from now on.
- 8 Assume S did not evict $p^*(=3)$ before we see $p' (=2)$.



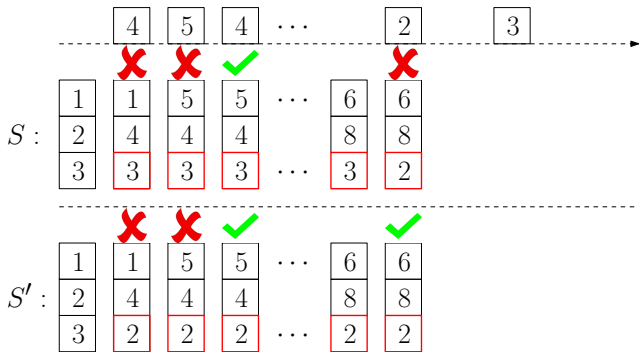
Proof.



Proof.

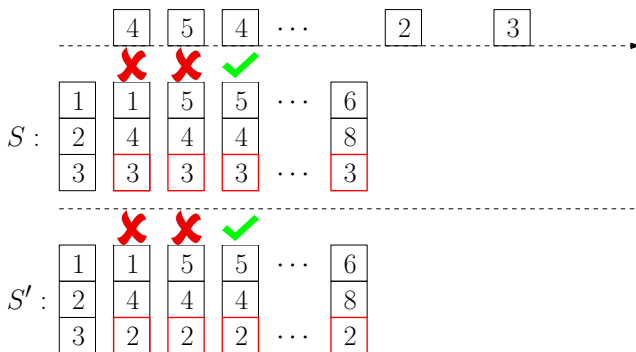


Proof.



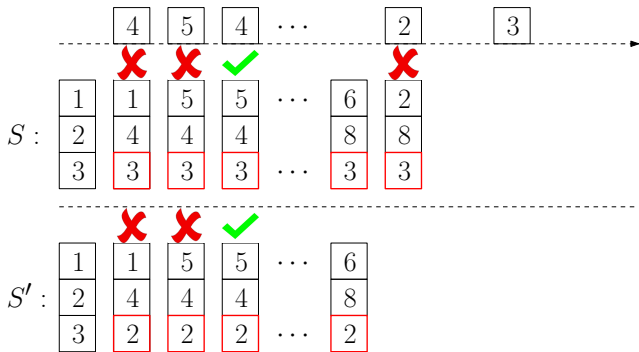
Proof.

- 9 If S evicts $p^*(=3)$ for $p' (=2)$, then S won't be optimum. Assume otherwise.



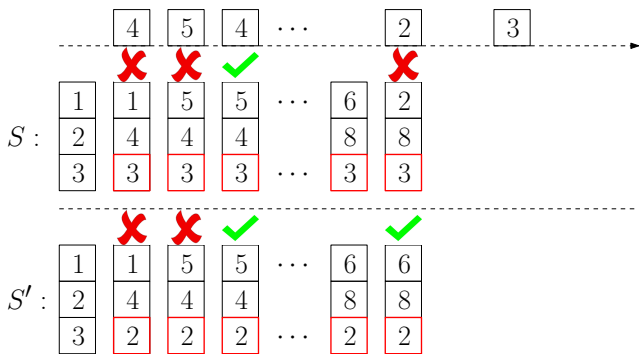
Proof.

- 9 If S evicts $p^*(=3)$ for $p'(=2)$, then S won't be optimum. Assume otherwise.



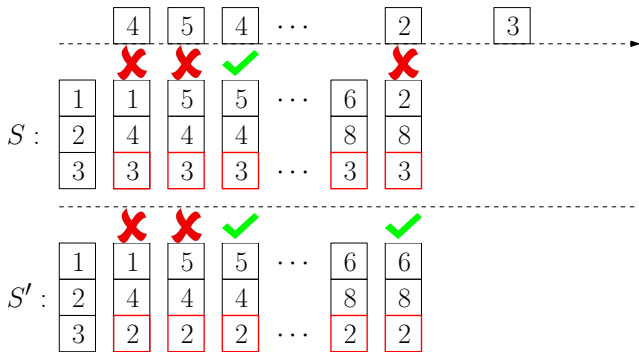
Proof.

- 9 If S evicts $p^*(=3)$ for $p'(=2)$, then S won't be optimum. Assume otherwise.



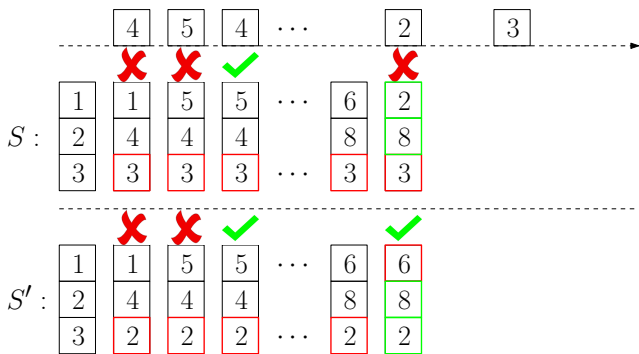
Proof.

- 9 If S evicts $p^*(=3)$ for $p'(=2)$, then S won't be optimum. Assume otherwise.



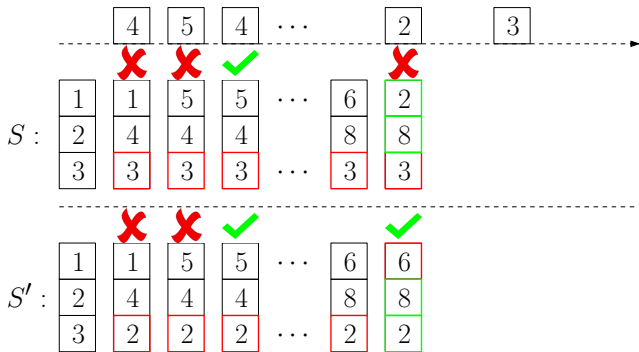
Proof.

- 9 If S evicts $p^*(=3)$ for $p'(=2)$, then S won't be optimum. Assume otherwise.
- 10 So far, S' has 1 less page-miss than S does.

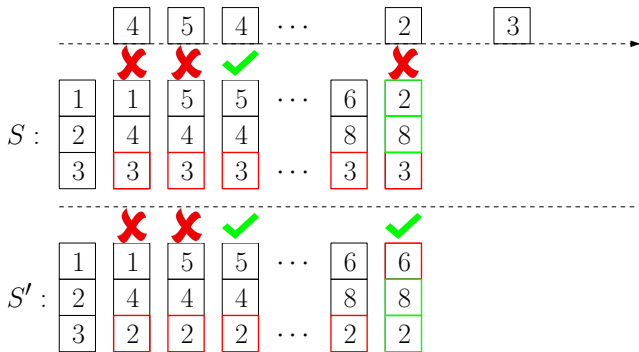


Proof.

- 9 If S evicts $p^*(=3)$ for $p'(=2)$, then S won't be optimum. Assume otherwise.
- 10 So far, S' has 1 less page-miss than S does.
- 11 The status of S' and that of S only differ by 1 page.

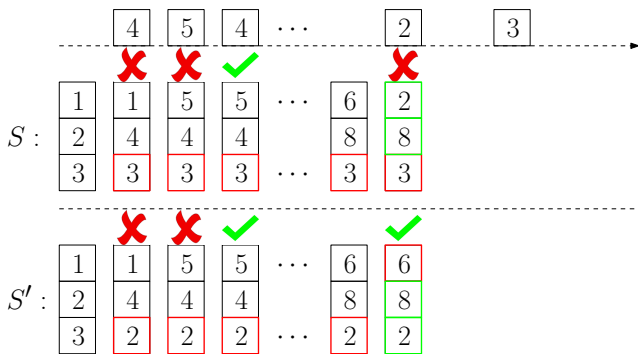


Proof.



Proof.

- 12 We can then guarantee that S' make at most the same number of page-misses as S does.



Proof.

- 12 We can then guarantee that S' make at most the same number of page-misses as S does.
- Idea: if S has a page-hit and S' has a page-miss, we use the opportunity to make the status of S' the same as that of S . □

- Thus, we have shown how to create another solution S' with the same number of page-misses as that of the optimum solution S . Thus, we proved

Lemma Assume at time 1 a page fault happens and there are no empty pages in the cache. Let p^* be the page in cache that is not requested until furthest in the future. **There is an optimum solution in which p^* is evicted at time 1.**

- Thus, we have shown how to create another solution S' with the same number of page-misses as that of the optimum solution S . Thus, we proved

Lemma Assume at time 1 a page fault happens and there are no empty pages in the cache. Let p^* be the page in cache that is not requested until furthest in the future. **It is safe to evict p^* at time 1.**

- Thus, we have shown how to create another solution S' with the same number of page-misses as that of the optimum solution S . Thus, we proved

Lemma Assume at time 1 a page fault happens and there are no empty pages in the cache. Let p^* be the page in cache that is not requested until furthest in the future. **It is safe to evict p^* at time 1.**

Theorem The furthest-in-future strategy is optimum.

```
1: for  $t \leftarrow 1$  to  $T$  do
2:   if  $\rho_t$  is in cache then do nothing
3:   else if there is an empty page in cache then
4:     evict the empty page and load  $\rho_t$  in cache
5:   else
6:      $p^* \leftarrow$  page in cache that is not used furthest in the future
7:     evict  $p^*$  and load  $\rho_t$  in cache
```

Q: How can we make the algorithm as fast as possible?

A:

Q: How can we make the algorithm as fast as possible?

A:

- The running time can be made to be $O(n + T \log k)$.

Q: How can we make the algorithm as fast as possible?

A:

- The running time can be made to be $O(n + T \log k)$.
- For each page p , use a linked list (or an array with dynamic size) to store the time steps in which p is requested.

Q: How can we make the algorithm as fast as possible?

A:

- The running time can be made to be $O(n + T \log k)$.
- For each page p , use a linked list (or an array with dynamic size) to store the time steps in which p is requested.
- We can find the next time a page is requested easily.

Q: How can we make the algorithm as fast as possible?

A:

- The running time can be made to be $O(n + T \log k)$.
- For each page p , use a linked list (or an array with dynamic size) to store the time steps in which p is requested.
 - We can find the next time a page is requested easily.
- Use a priority queue data structure to hold all the pages in cache, so that we can easily find the page that is requested furthest in the future.

time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3

P1:

1	10
---	----

P2:

4	7
---	---

P3:

6	9	12
---	---	----

P4:

3	8
---	---

P5:

2	5	11
---	---	----

priority queue

pages	priority values

time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3

P1:

1	10
---	----

P2:

4	7
---	---

P3:

6	9	12
---	---	----

P4:

3	8
---	---

P5:

2	5	11
---	---	----

priority queue

pages	priority values



time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3

P1:

1	10
---	----

P2:

4	7
---	---

P3:

6	9	12
---	---	----

P4:

3	8
---	---

P5:

2	5	11
---	---	----

priority queue

pages	priority values



time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3



P1:

1	10
---	----

P2:

4	7
---	---

P3:

6	9	12
---	---	----

P4:

3	8
---	---

P5:

2	5	11
---	---	----

priority queue

pages	priority values
P1	10

time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3

X

P1: 1 10

P2: 4 7

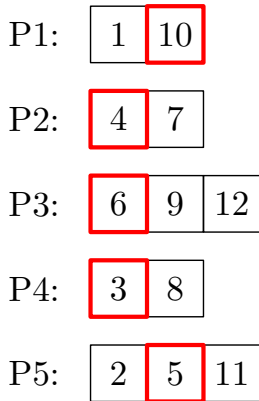
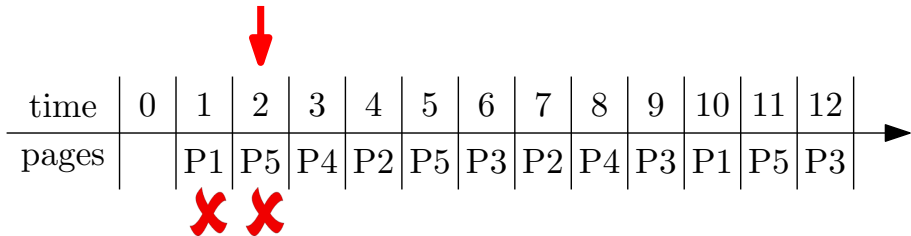
P3: 6 9 12

P4: 3 8

P5: 2 5 11

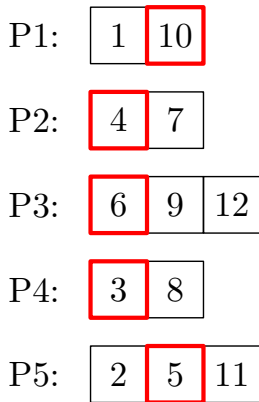
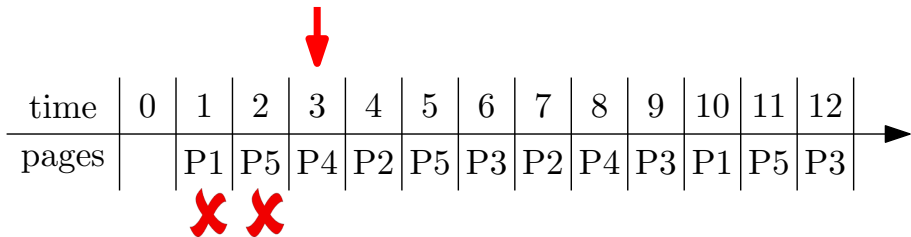
priority queue

pages	priority values
P1	10



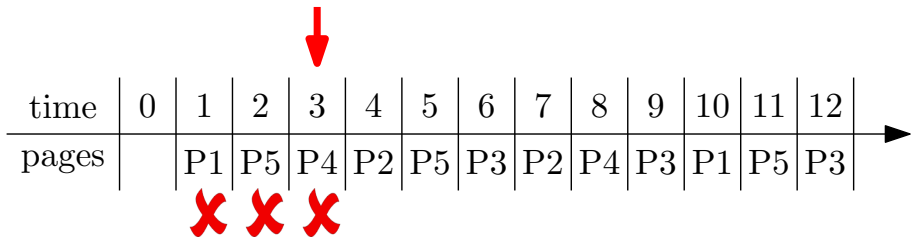
priority queue

pages	priority values
P1	10
P5	5



priority queue

pages	priority values
P1	10
P5	5



P1:

1	10
---	----

P2:

4	7
---	---

P3:

6	9	12
---	---	----

P4:

3	8
---	---

P5:

2	5	11
---	---	----

priority queue

pages	priority values
P1	10
P5	5
P4	8

time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3

↓

X X X

P1:

1	10
---	----

P2:

4	7
---	---

P3:

6	9	12
---	---	----

P4:

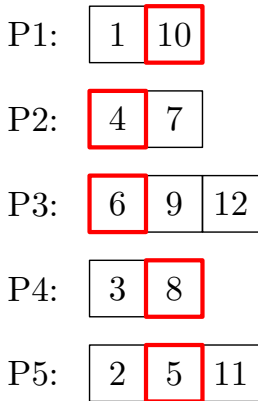
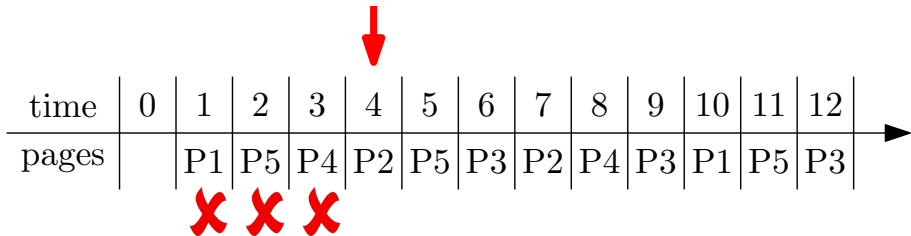
3	8
---	---

P5:

2	5	11
---	---	----

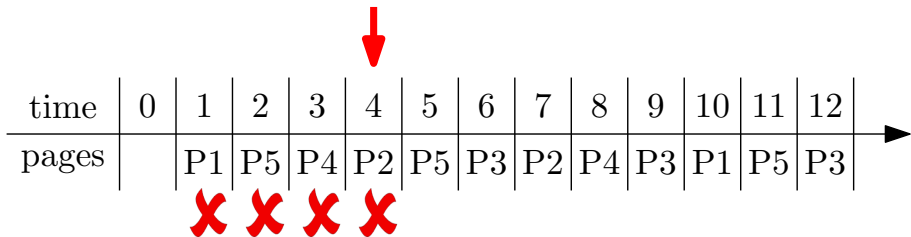
priority queue

pages	priority values
P1	10
P5	5
P4	8



priority queue

pages	priority values
P5	5
P4	8



P1:

1	10
---	----

P2:

4	7
---	---

P3:

6	9	12
---	---	----

P4:

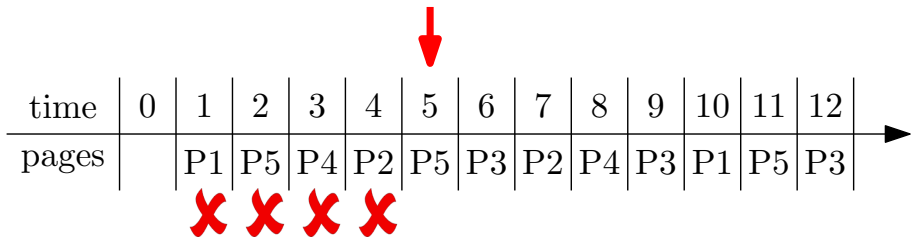
3	8
---	---

P5:

2	5	11
---	---	----

priority queue

pages	priority values
P2	7
P5	5
P4	8



P1:

1	10
---	----

P2:

4	7
---	---

P3:

6	9	12
---	---	----

P4:

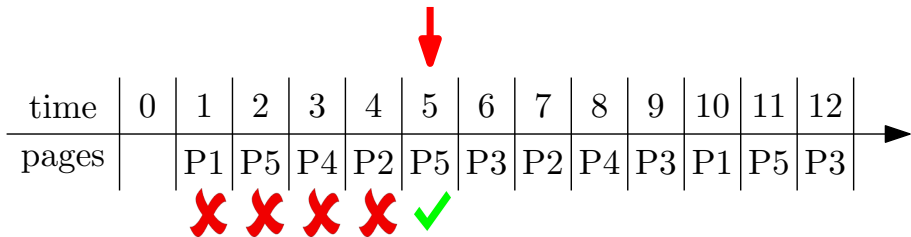
3	8
---	---

P5:

2	5	11
---	---	----

priority queue

pages	priority values
P2	7
P5	5
P4	8



P1:

1	10
---	----

P2:

4	7
---	---

P3:

6	9	12
---	---	----

P4:

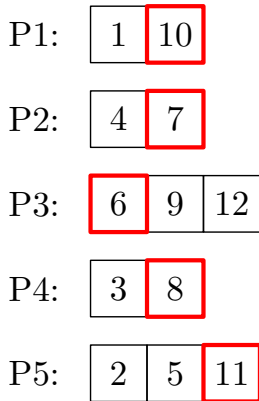
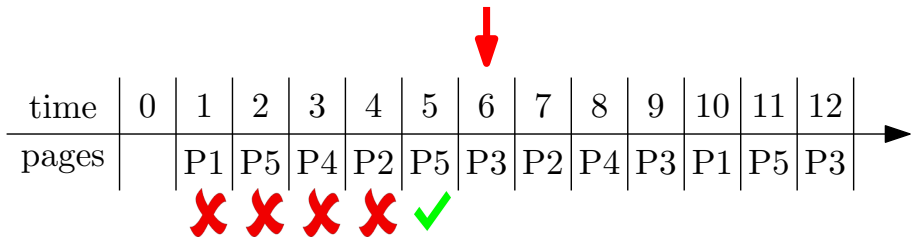
3	8
---	---

P5:

2	5	11
---	---	----

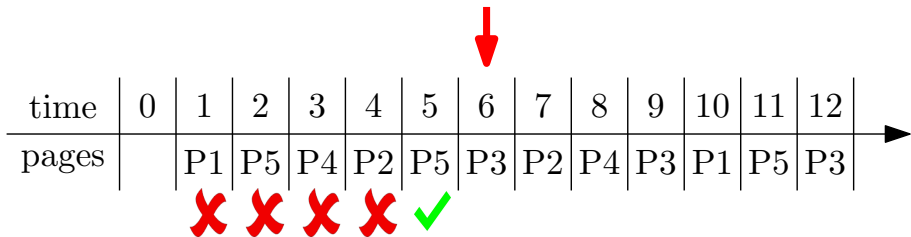
priority queue

pages	priority values
P2	7
P5	11
P4	8



priority queue

pages	priority values
P2	7
P5	11
P4	8



P1:

1	10
---	----

P2:

4	7
---	---

P3:

6	9	12
---	---	----

P4:

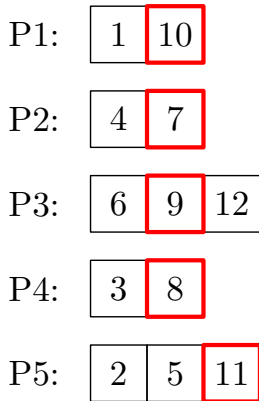
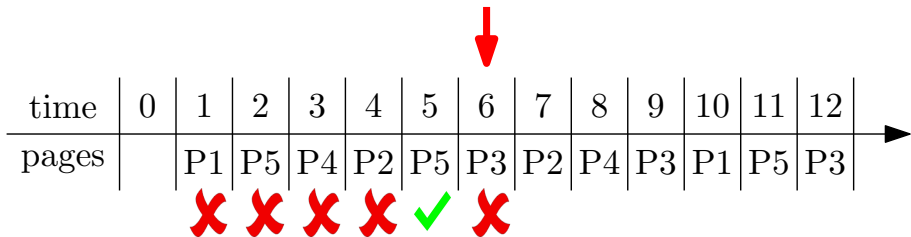
3	8
---	---

P5:

2	5	11
---	---	----

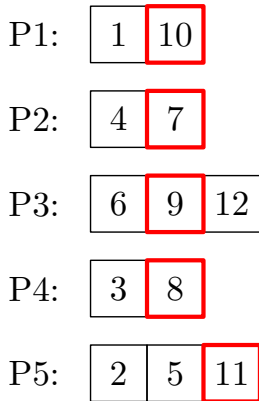
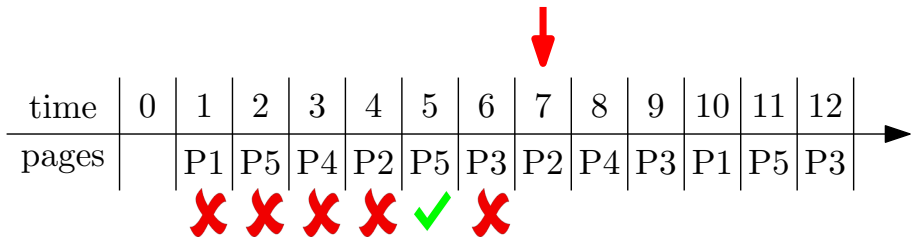
priority queue

pages	priority values
P2	7
P4	8



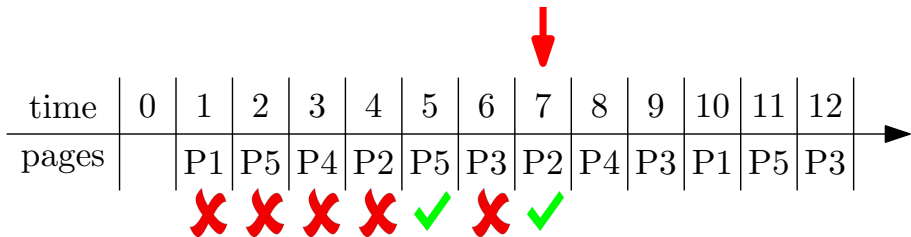
priority queue

pages	priority values
P2	7
P3	9
P4	8



priority queue

pages	priority values
P2	7
P3	9
P4	8



P1:

1	10
---	----

P2:

4	7	
---	---	--

P3:

6	9	12
---	---	----

P4:

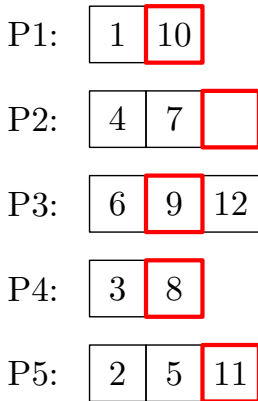
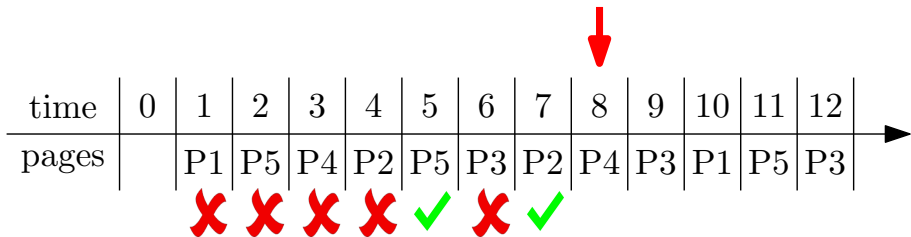
3	8
---	---

P5:

2	5	11
---	---	----

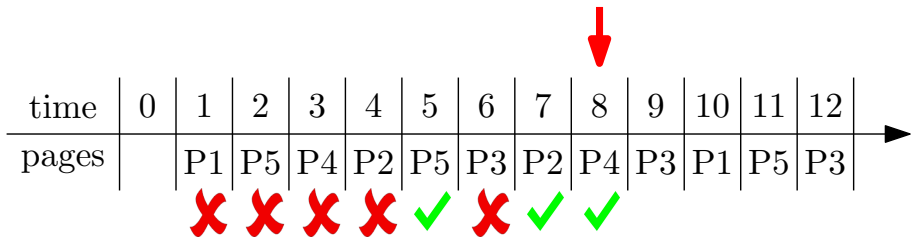
priority queue

pages	priority values
P2	∞
P3	9
P4	8



priority queue

pages	priority values
P2	∞
P3	9
P4	8



P1:

1	10
---	----

P2:

4	7	
---	---	--

P3:

6	9	12
---	---	----

P4:

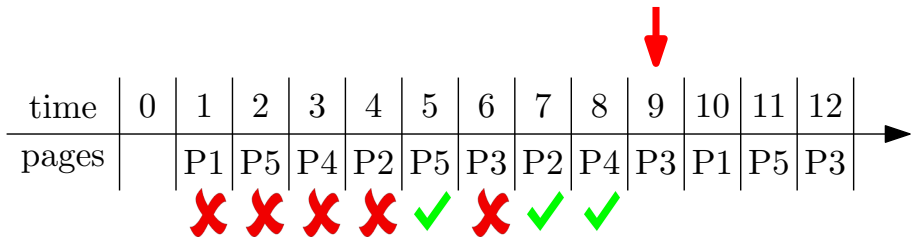
3	8	
---	---	--

P5:

2	5	11
---	---	----

priority queue

pages	priority values
P2	∞
P3	9
P4	∞



P1:

1	10
---	----

P2:

4	7	
---	---	--

P3:

6	9	12
---	---	----

P4:

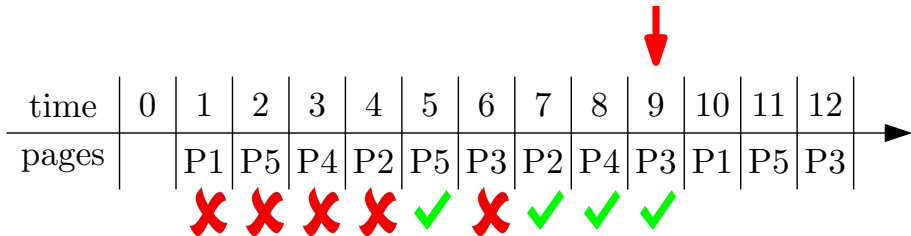
3	8	
---	---	--

P5:

2	5	11
---	---	----

priority queue

pages	priority values
P2	∞
P3	9
P4	∞



P1: 1 10

P2: 4 7

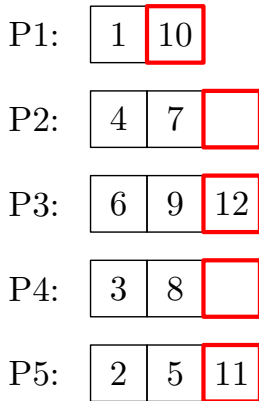
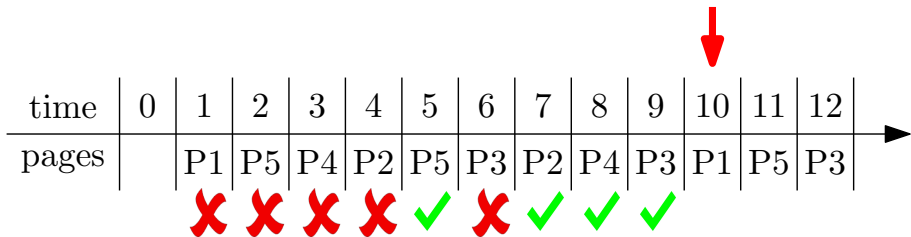
P3: 6 9 12

P4: 3 8

P5: 2 5 11

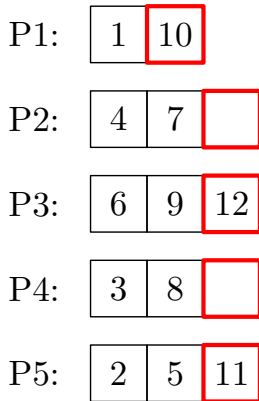
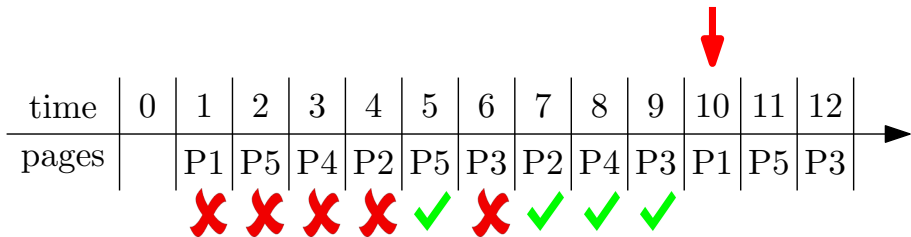
priority queue

pages	priority values
P2	∞
P3	12
P4	∞



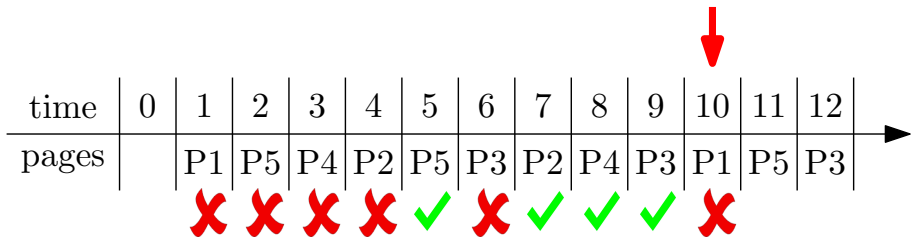
priority queue

pages	priority values
P2	∞
P3	12
P4	∞



priority queue

pages	priority values
P3	12
P4	∞



P1:

1	10	
---	----	--

P2:

4	7	
---	---	--

P3:

6	9	12
---	---	----

P4:

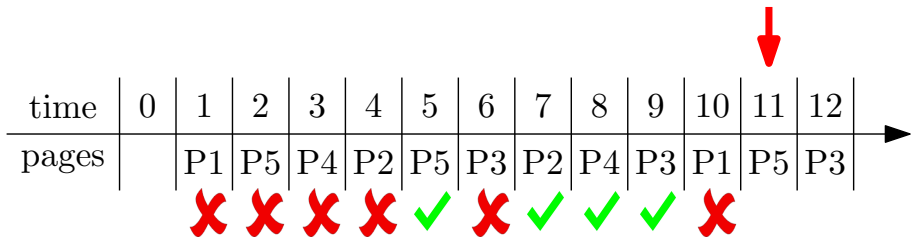
3	8	
---	---	--

P5:

2	5	11
---	---	----

priority queue

pages	priority values
P1	∞
P3	12
P4	∞



P1: 1 10

P2: 4 7

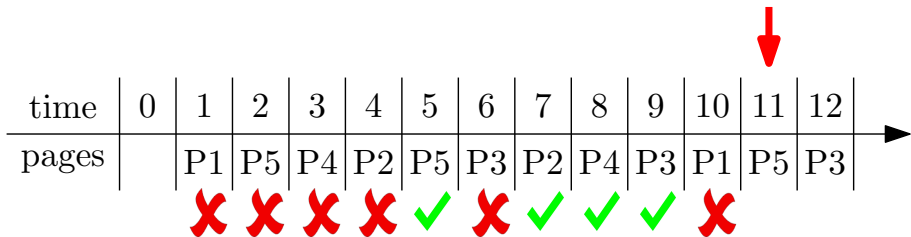
P3: 6 9 12

P4: 3 8

P5: 2 5 11

priority queue

pages	priority values
P1	∞
P3	12
P4	∞



P1: 1 | 10 |

P2: 4 | 7 |

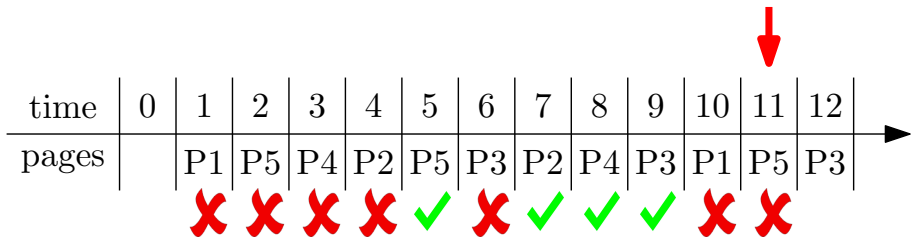
P3: 6 | 9 | 12

P4: 3 | 8 |

P5: 2 | 5 | 11

priority queue

pages	priority values
P3	12
P4	∞



P1: 1 | 10 |

P2: 4 | 7 |

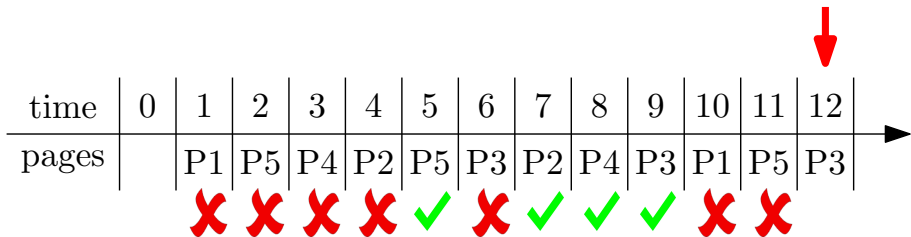
P3: 6 | 9 |

P4: 3 | 8 |

P5: 2 | 5 | 11 |

priority queue

pages	priority values
P5	∞
P3	12
P4	∞



P1: 1 10

P2: 4 7

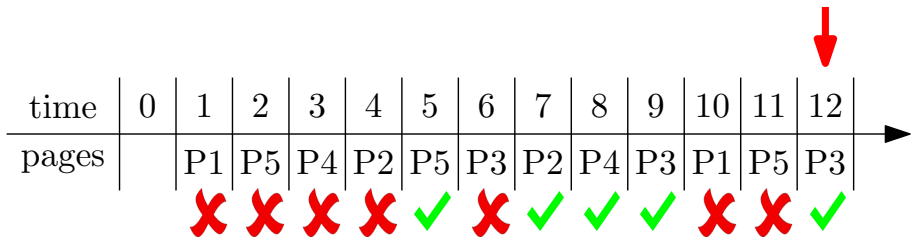
P3: 6 9 12

P4: 3 8

P5: 2 5 11

priority queue

pages	priority values
P5	∞
P3	12
P4	∞



P1: 1 10

P2: 4 7

P3: 6 9 12

P4: 3 8

P5: 2 5 11

priority queue

pages	priority values
P5	∞
P3	∞
P4	∞

```

1: for every  $p \leftarrow 1$  to  $n$  do
2:    $times[p] \leftarrow$  array of times in which  $p$  is requested, in
   increasing order                                ▷ put  $\infty$  at the end of array
3:    $pointer[p] \leftarrow 1$ 
4:  $Q \leftarrow$  empty priority queue
5: for every  $t \leftarrow 1$  to  $T$  do
6:    $pointer[\rho_t] \leftarrow pointer[\rho_t] + 1$ 
7:   if  $\rho_t \in Q$  then
8:      $Q.increase\text{-}key(\rho_t, times[\rho_t, pointer[\rho_t]])$ , print "hit",
continue
9:   if  $Q.size() < k$  then
10:    print "load  $\rho_t$  to an empty page "
11:   else
12:     $p \leftarrow Q.extract\text{-}max()$ , print "evict  $p$  and load  $\rho_t$ "
13:     $Q.insert(\rho_t, times[\rho_t, pointer[\rho_t]])$     ▷ add  $\rho_t$  to  $Q$  with key
value  $times[\rho_t, pointer[\rho_t]]$ 

```

Outline

- 1 Toy Example: Box Packing
- 2 Interval Scheduling
 - Interval Partitioning
- 3 Offline Caching**
 - Heap: Concrete Data Structure for Priority Queue
- 4 Data Compression and Huffman Code
- 5 Summary
- 6 Exercise Problems

- Let V be a ground set of size n .

Def. A **priority queue** is an **abstract** data structure that maintains a set $U \subseteq V$ of elements, each with an associated key value, and supports the following operations:

- $\text{insert}(v, \text{key_value})$: insert an element $v \in V \setminus U$, with associated key value key_value .
- $\text{decrease_key}(v, \text{new_key_value})$: decrease the key value of an element $v \in U$ to new_key_value
- $\text{extract_min}()$: return and remove the element in U with the smallest key value
- ...

Simple Implementations for Priority Queue

- n = size of ground set V

data structures	insert	extract_min	decrease_key
array			
sorted array			

Simple Implementations for Priority Queue

- $n =$ size of ground set V

data structures	insert	extract_min	decrease_key
array	$O(1)$	$O(n)$	$O(1)$
sorted array			

Simple Implementations for Priority Queue

- $n =$ size of ground set V

data structures	insert	extract_min	decrease_key
array	$O(1)$	$O(n)$	$O(1)$
sorted array	$O(n)$	$O(1)$	$O(n)$

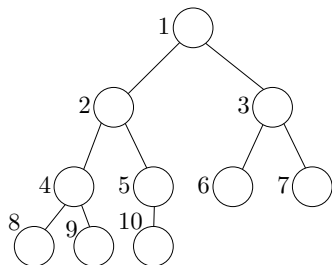
Simple Implementations for Priority Queue

- $n =$ size of ground set V

data structures	insert	extract_min	decrease_key
array	$O(1)$	$O(n)$	$O(1)$
sorted array	$O(n)$	$O(1)$	$O(n)$
heap	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$

Heap

The elements in a heap is organized using a complete binary tree:

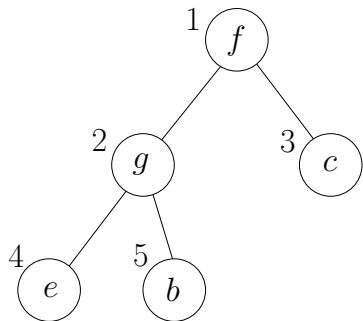


- Nodes are indexed as $\{1, 2, 3, \dots, s\}$
- Parent of node i : $\lfloor i/2 \rfloor$
- Left child of node i : $2i$
- Right child of node i : $2i + 1$

Heap

A heap H contains the following fields

- s : size of U (number of elements in the heap)
- $A[i], 1 \leq i \leq s$: the element at node i of the tree
- $p[v], v \in U$: the index of node containing v
- $key[v], v \in U$: the key value of element v

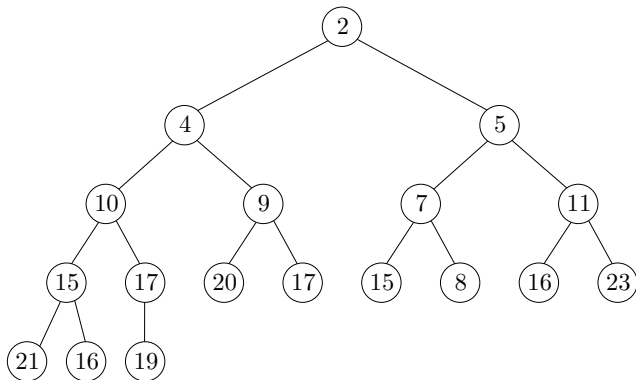


- $s = 5$
- $A = ('f', 'g', 'c', 'e', 'b')$
- $p['f'] = 1, p['g'] = 2, p['c'] = 3,$
 $p['e'] = 4, p['b'] = 5$

Heap

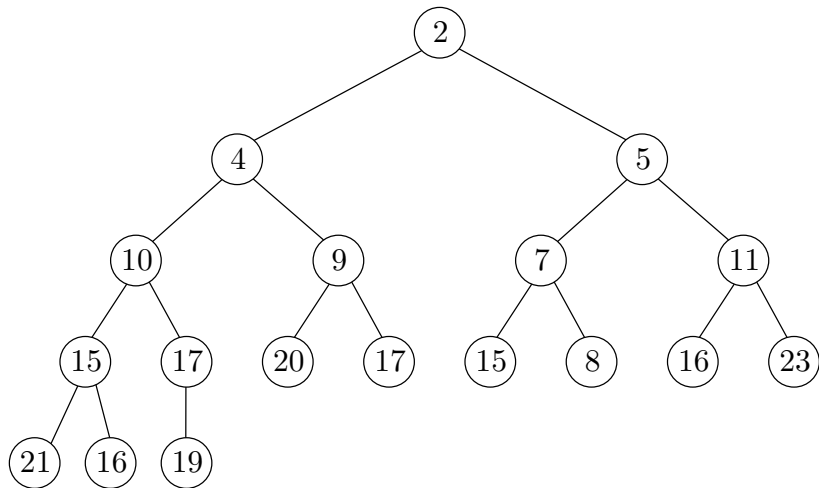
The following **heap property** is satisfied:

- for any two nodes i, j such that i is the parent of j , we have $key[A[i]] \leq key[A[j]]$.

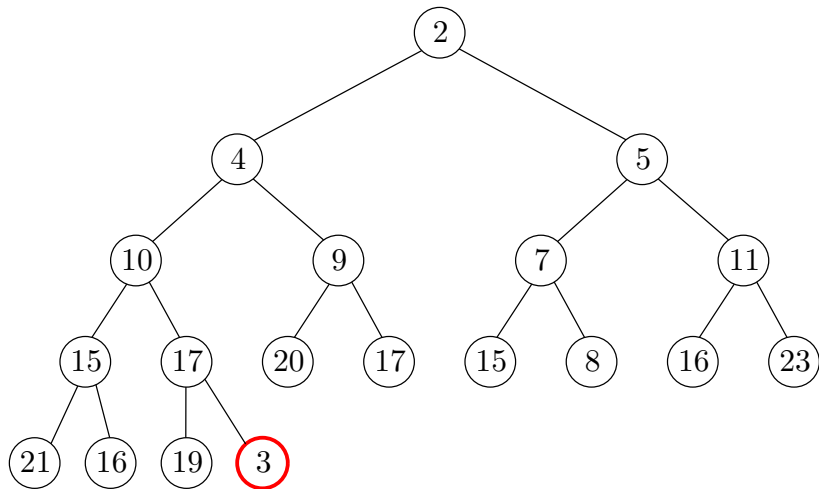


A heap. Numbers in the circles denote key values of elements.

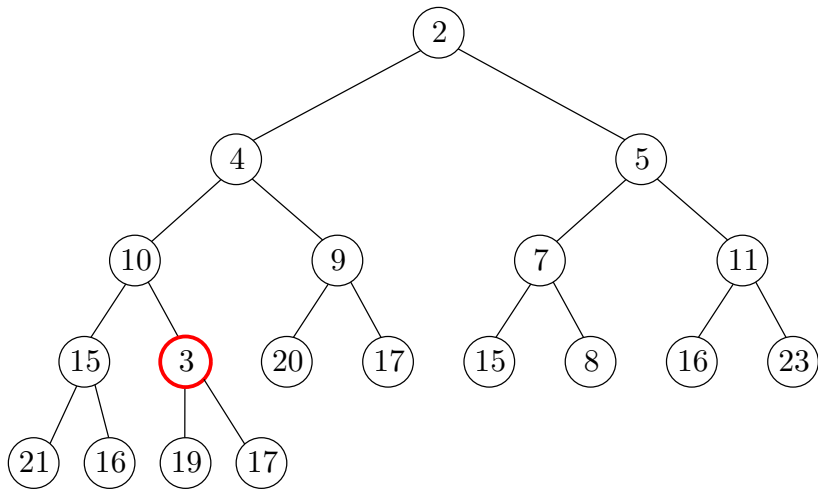
$\text{insert}(v, \text{key_value})$



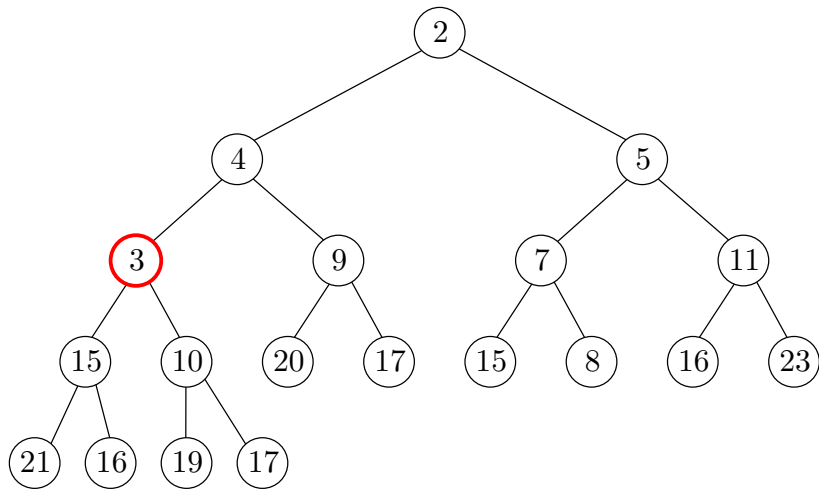
$\text{insert}(v, \text{key_value})$



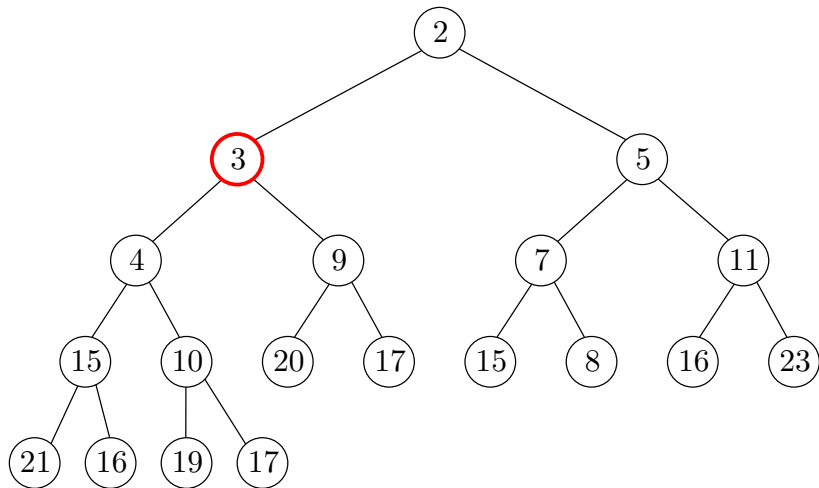
$\text{insert}(v, \text{key_value})$



$\text{insert}(v, \text{key_value})$



$\text{insert}(v, \text{key_value})$



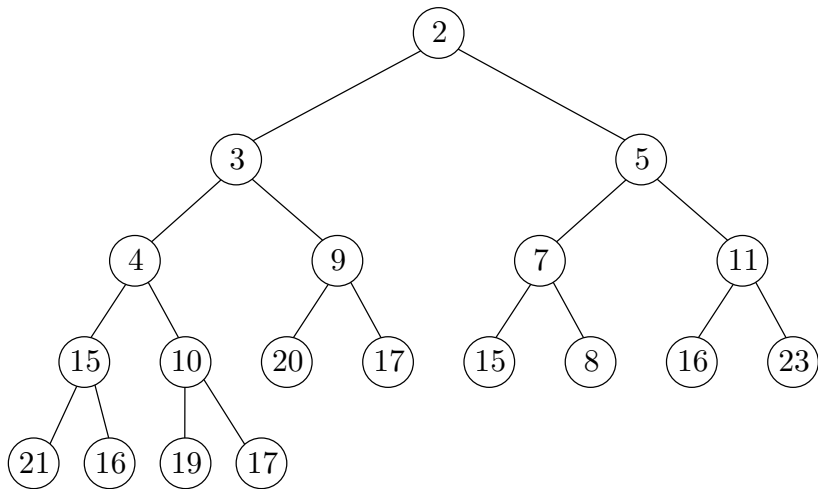
insert(v , key_value)

```
1:  $s \leftarrow s + 1$   
2:  $A[s] \leftarrow v$   
3:  $p[v] \leftarrow s$   
4:  $key[v] \leftarrow key\_value$   
5: heapify-up( $s$ )
```

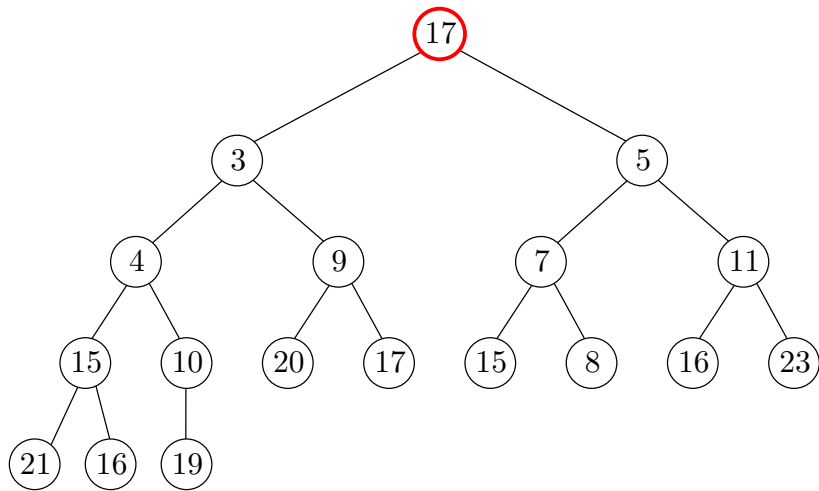
heapify-up(i)

```
1: while  $i > 1$  do  
2:    $j \leftarrow \lfloor i/2 \rfloor$   
3:   if  $key[A[i]] < key[A[j]]$  then  
4:     swap  $A[i]$  and  $A[j]$   
5:      $p[A[i]] \leftarrow i, p[A[j]] \leftarrow j$   
6:      $i \leftarrow j$   
7:   else break
```

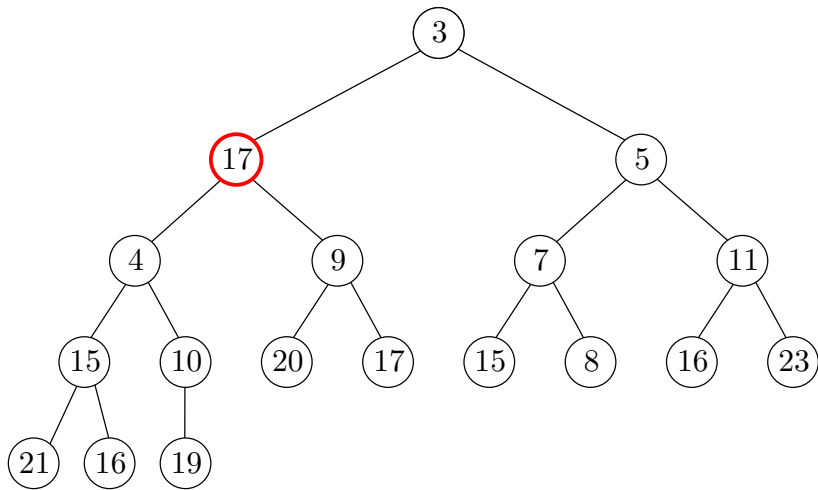
`extract_min()`



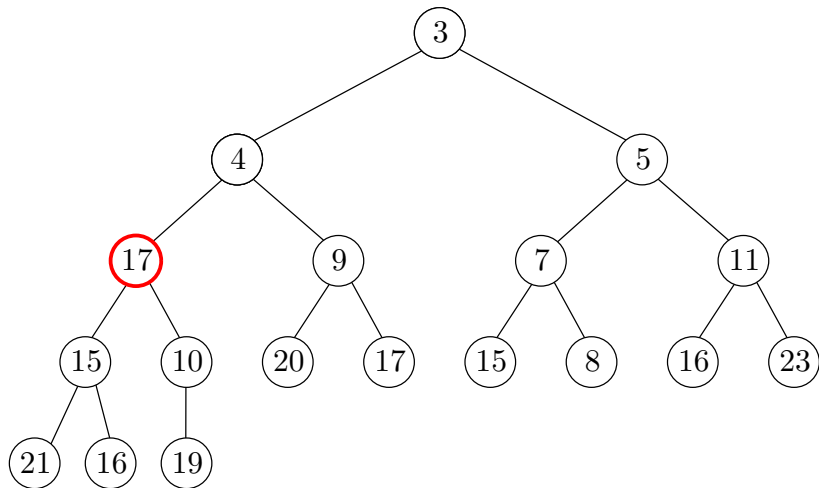
extract_min()



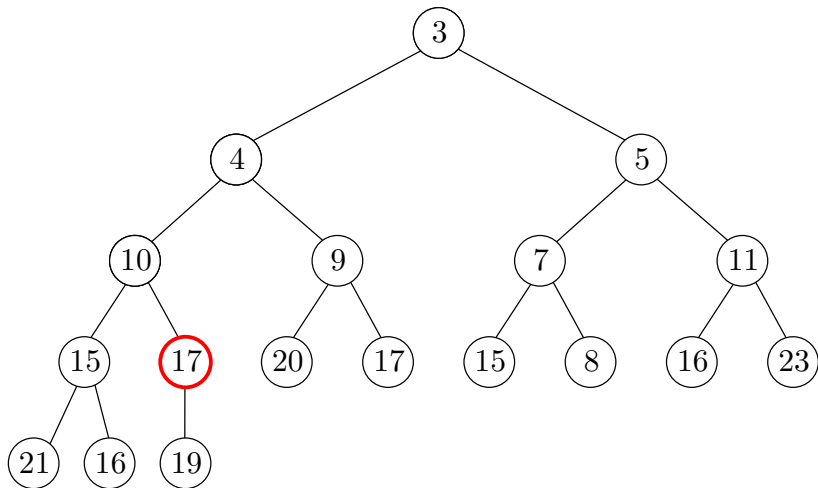
extract_min()



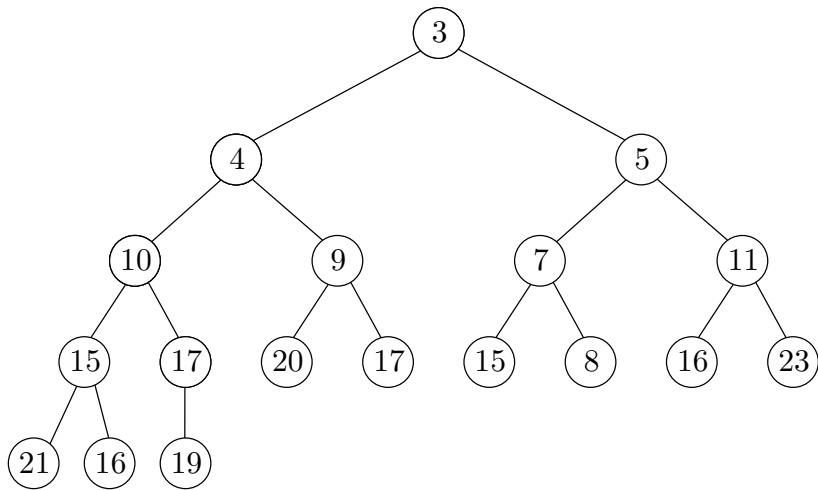
extract_min()



`extract_min()`



`extract_min()`



extract_min()

```
1:  $ret \leftarrow A[1]$ 
2:  $A[1] \leftarrow A[s]$ 
3:  $p[A[1]] \leftarrow 1$ 
4:  $s \leftarrow s - 1$ 
5: if  $s \geq 1$  then
6:   heapify_down(1)
7: return  $ret$ 
```

decrease_key(v, key_val)

```
1:  $key[v] \leftarrow key\_value$ 
2: heapify-up( $p[v]$ )
```

heapify-down(i)

```
1: while  $2i \leq s$  do
2:   if  $2i = s$  or
    $key[A[2i]] \leq key[A[2i + 1]]$  then
3:      $j \leftarrow 2i$ 
4:   else
5:      $j \leftarrow 2i + 1$ 
6:   if  $key[A[j]] < key[A[i]]$  then
7:     swap  $A[i]$  and  $A[j]$ 
8:      $p[A[i]] \leftarrow i, p[A[j]] \leftarrow j$ 
9:      $i \leftarrow j$ 
10:  else break
```