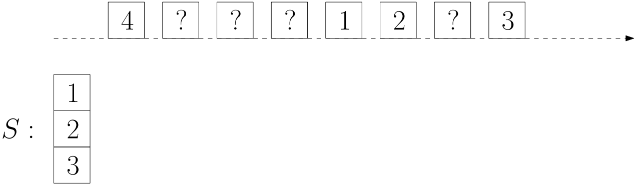
Analysis of Greedy Algorithm

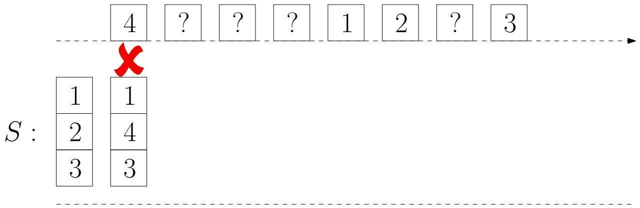
- Safety: Prove that the reasonable strategy is “safe” (key)
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

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.



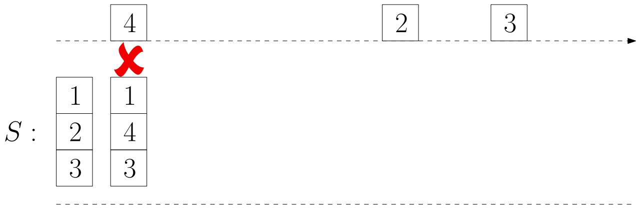
Proof.

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



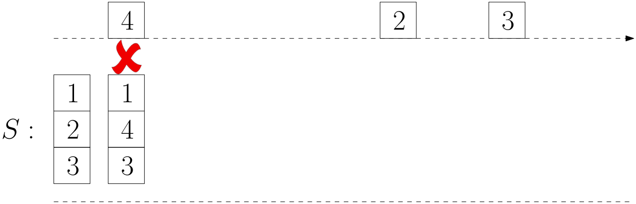
Proof.

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, $p' = 2$.

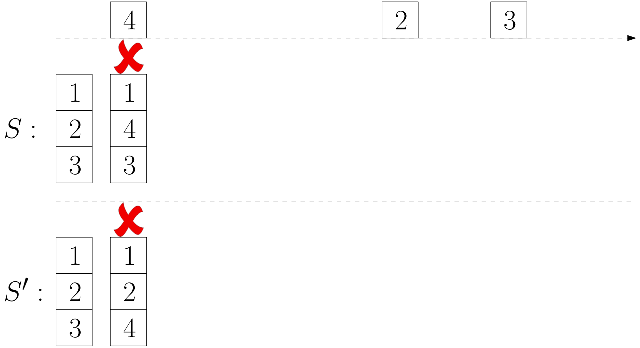


Proof.

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, $p' = 2$.

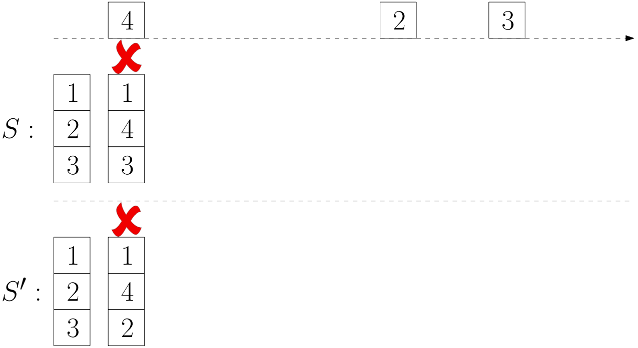


Proof.



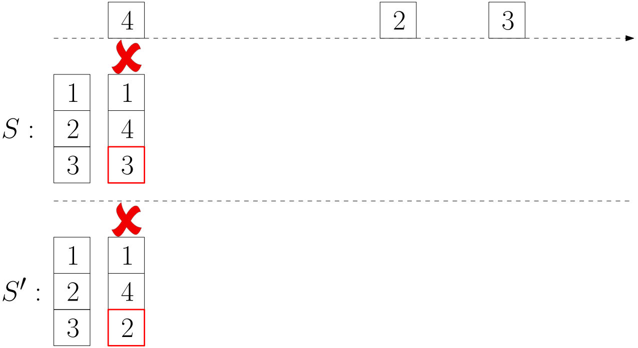
Proof.

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



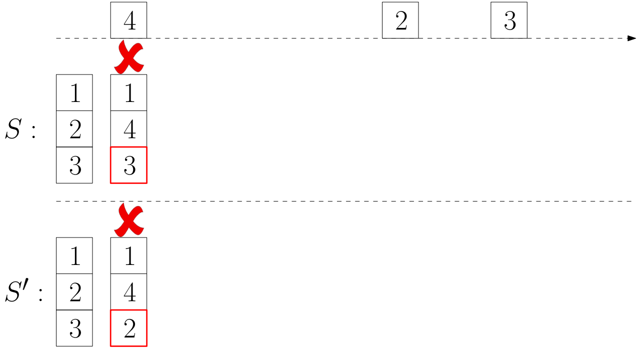
Proof.

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



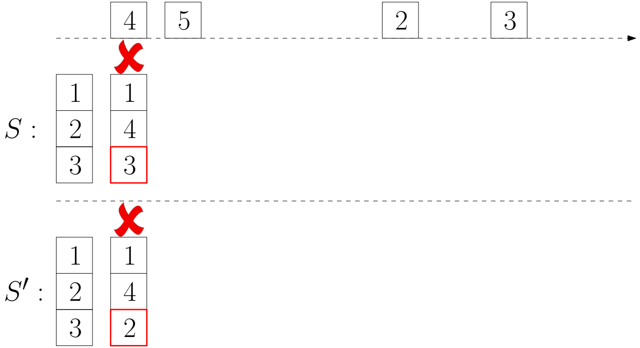
Proof.

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 $p^*(=3)$.



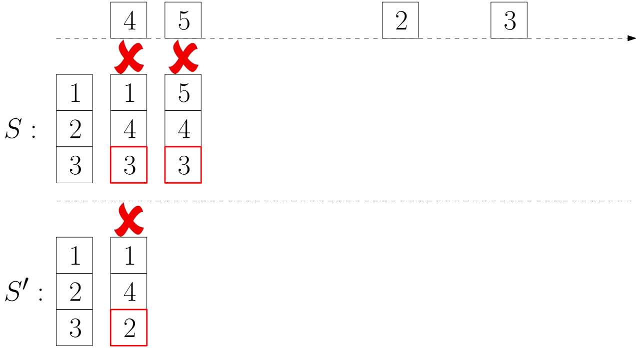
Proof.

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.



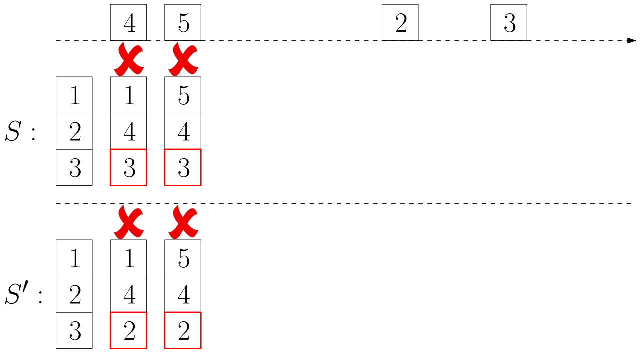
Proof.

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.



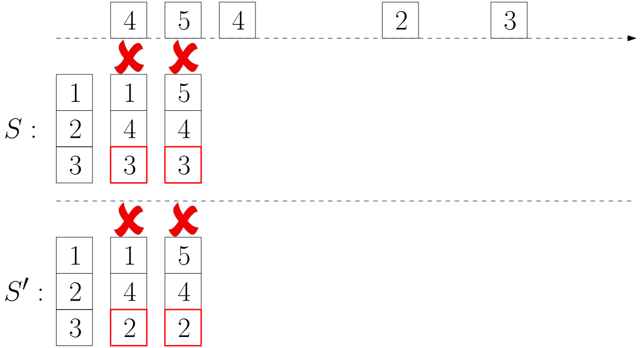
Proof.

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.



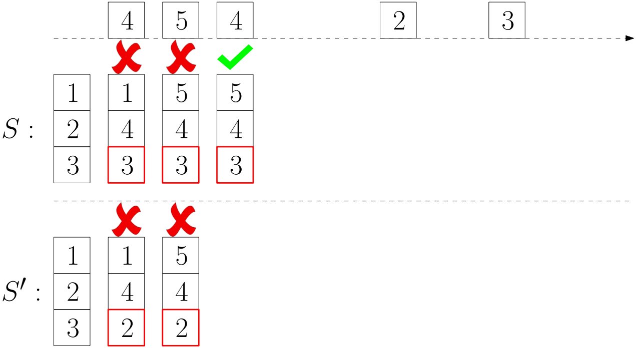
Proof.

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.



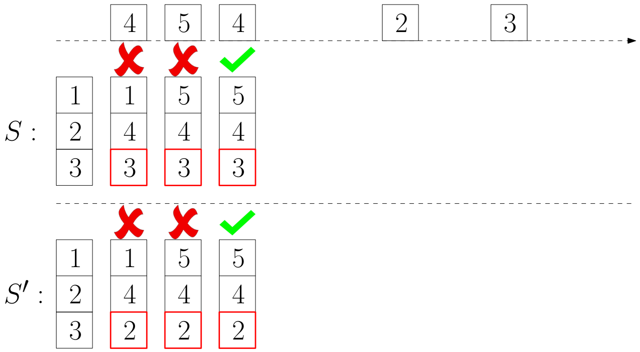
Proof.

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.



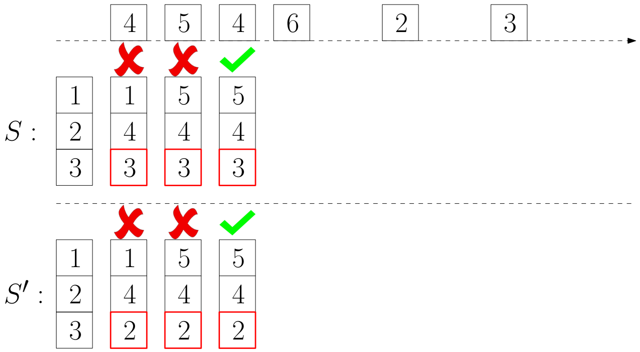
Proof.

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.



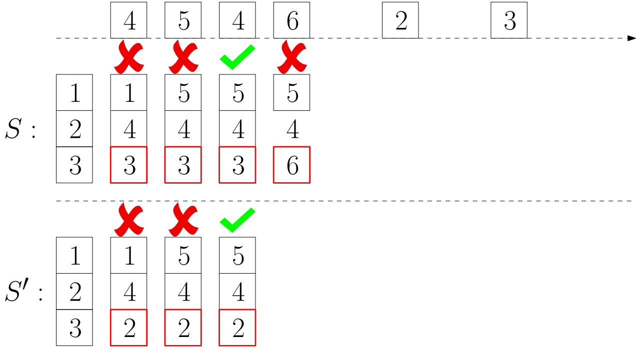
Proof.

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.



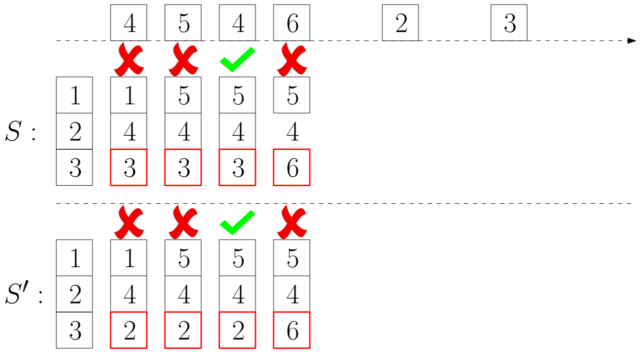
Proof.

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



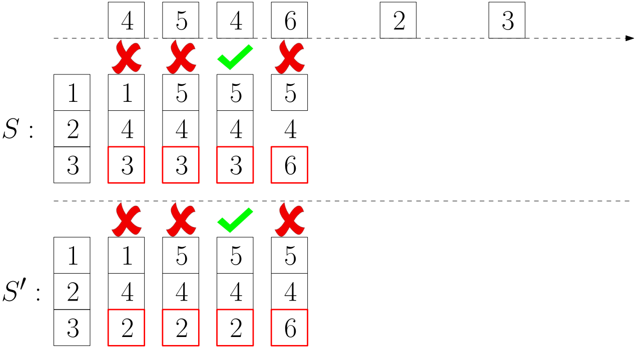
Proof.

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.

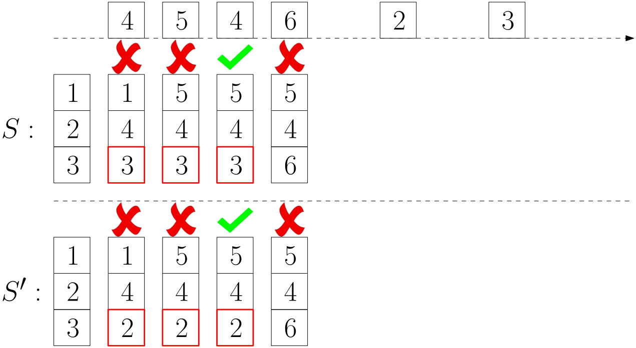


Proof.

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.

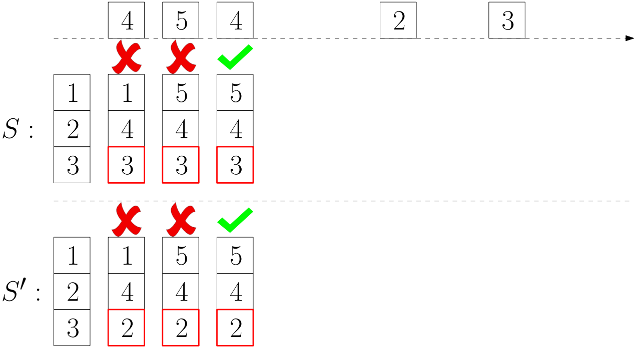


Proof.



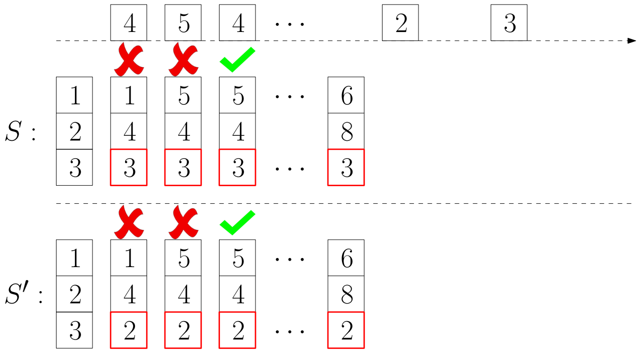
Proof.

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.



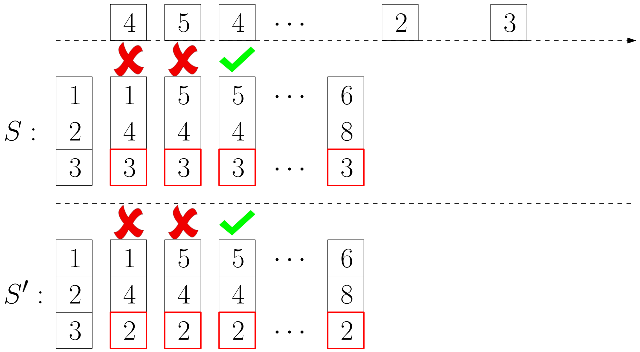
Proof.

1. 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.
2. Assume S did not evict $p^*(=3)$ before we see $p'(=2)$.

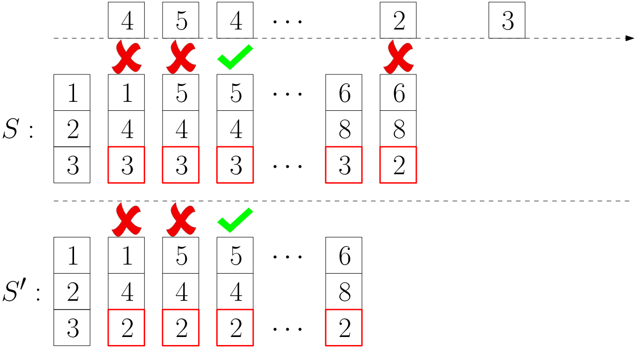


Proof.

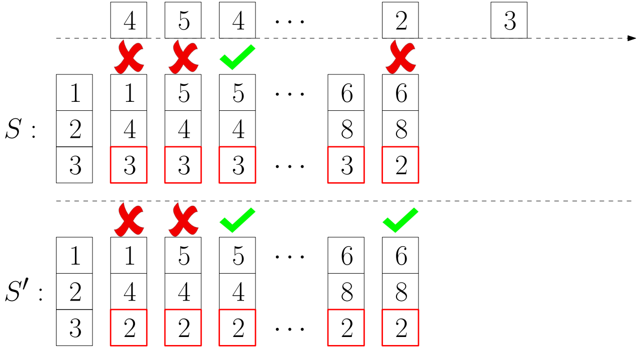
7. If S evicted the page p^*, S' will evict the page p'. Then, the cache status of S and that of S' will be the same. S and S' will be exactly the same from now on.
8. Assume S did not evict $p^*(=3)$ before we see $p'(=2)$.



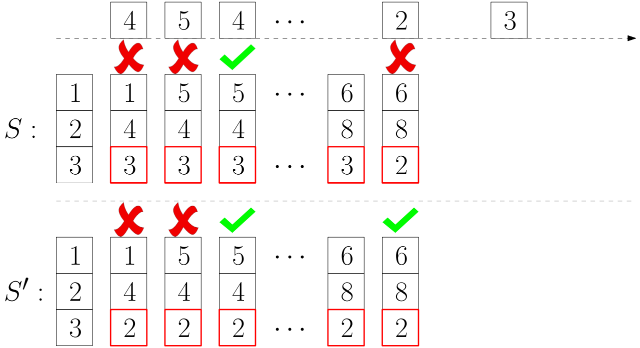
Proof.



Proof.

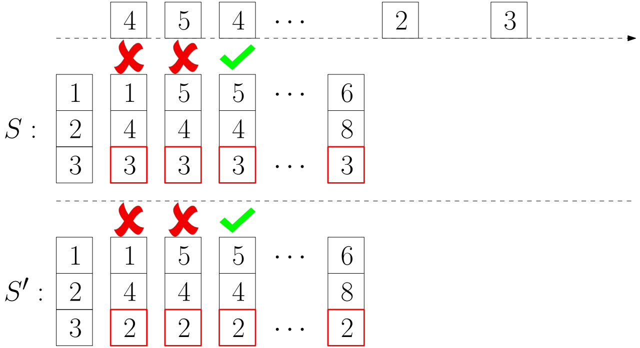


Proof.



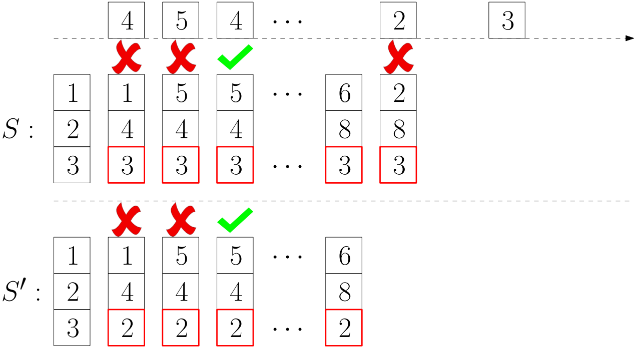
Proof.

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



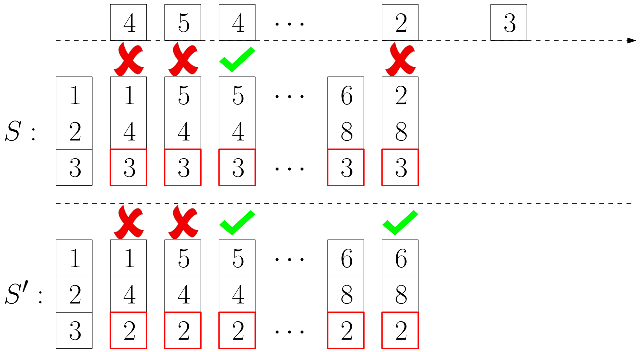
Proof.

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



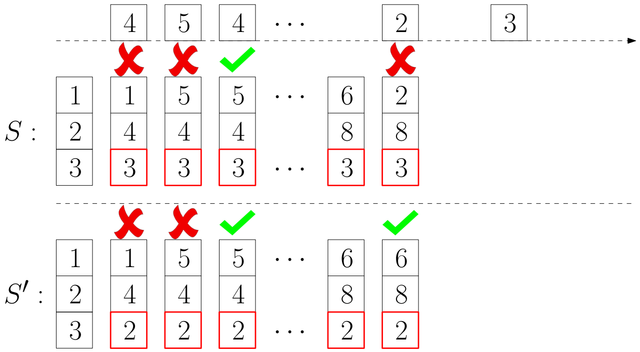
Proof.

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



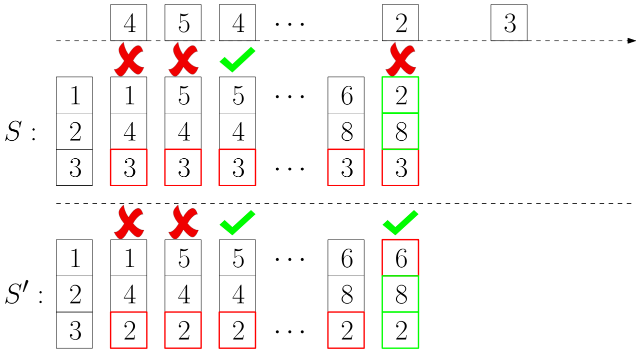
Proof.

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



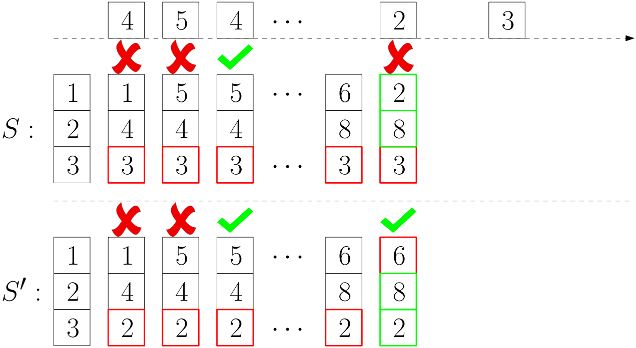
Proof.

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.

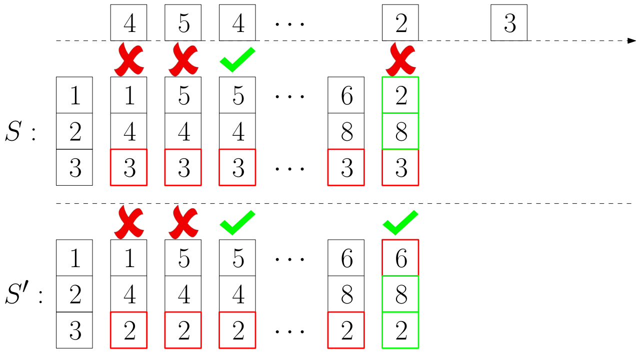


Proof.

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.

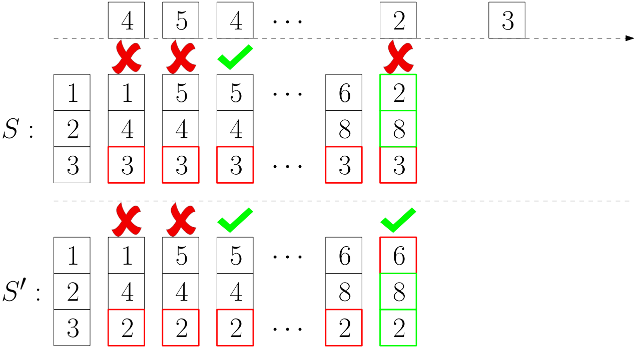


Proof.



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.

for $t \leftarrow 1$ to T **do**
    if ρ_t is in cache **then** do nothing
    else if there is an empty page in cache **then**
        evict the empty page and load ρ_t in cache
    else
        $p^* \leftarrow$ page in cache that is not used furthest in the future
        evict p^* and load ρ_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

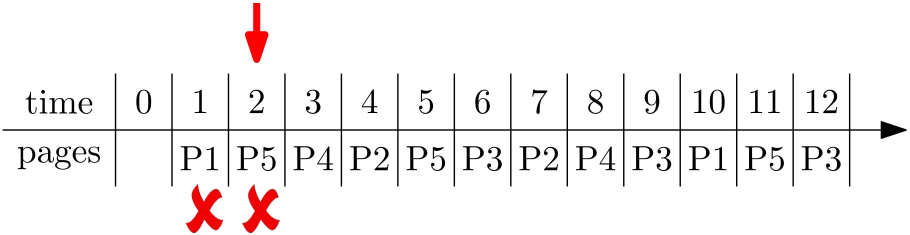
pages	priority values
P1	10

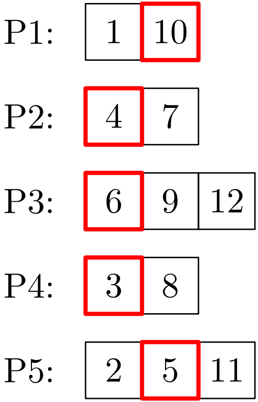
time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3

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

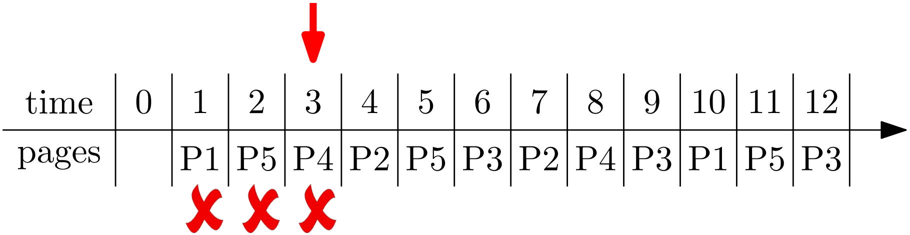
pages	priority values
P1	10
P5	5

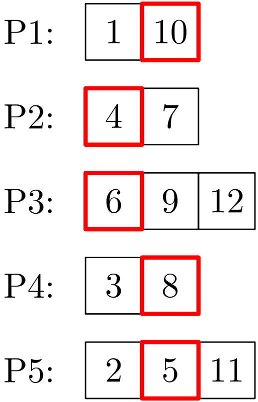
time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3

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

priority queue

pages	priority values
P1	10
P5	5





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

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

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

priority queue

pages	priority values
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

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

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
P2	7
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
		✗	✗	✗	✗	✓							

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

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

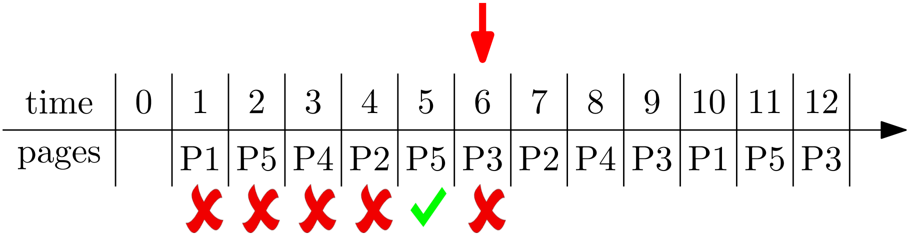
pages	priority values
P2	7
P5	11
P4	8

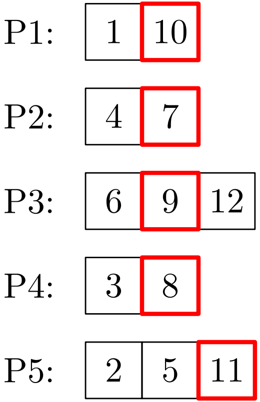
time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3

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

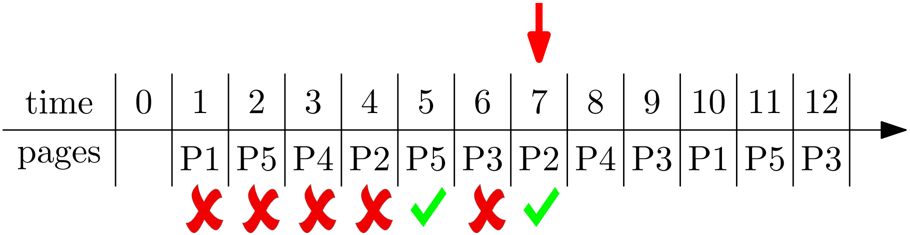
pages	priority values
P2	7
P3	9
P4	8

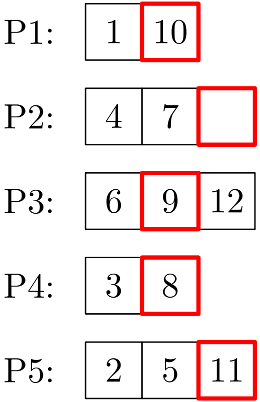
time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3

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

priority queue

pages	priority values
P2	7
P3	9
P4	8

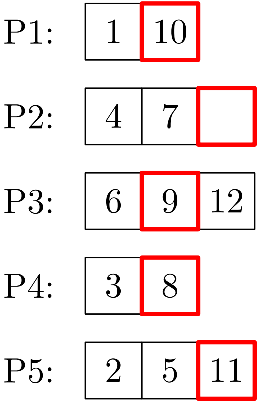




priority queue

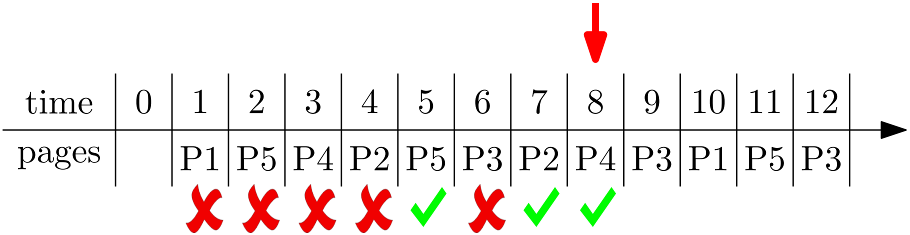
pages	priority values
P2	∞
P3	9
P4	8

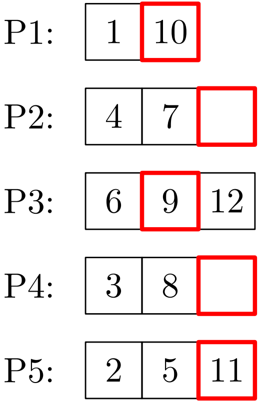
time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3
		✗	✗	✗	✗	✓	✗	✓					



priority queue

pages	priority values
P2	∞
P3	9
P4	8





priority queue

pages	priority values
P2	∞
P3	9
P4	∞

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
P2	∞
P3	9
P4	∞

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
P2	∞
P3	12
P4	∞

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
P2	∞
P3	12
P4	∞

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

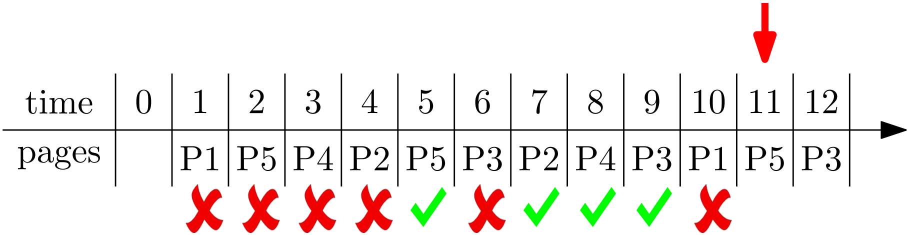
pages	priority values
P3	12
P4	∞

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	∞
P3	12
P4	∞



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

priority queue

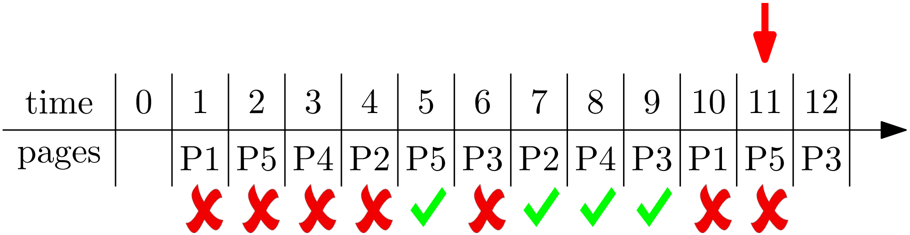
pages	priority values
P1	∞
P3	12
P4	∞

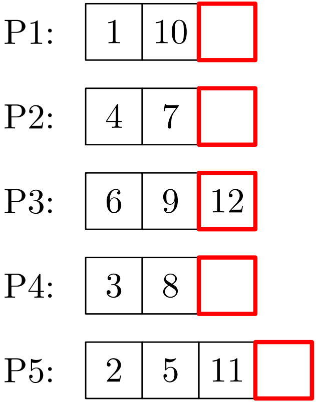
time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3
		✗	✗	✗	✗	✓	✗	✓	✓	✓	✗		

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

priority queue

pages	priority values
P3	12
P4	∞





priority queue

pages	priority values
P5	∞
P3	12
P4	∞

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
P5	∞
P3	12
P4	∞

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
P5	∞
P3	∞
P4	∞

for every $p \leftarrow 1$ to n **do**
    $times[p] \leftarrow$ array of times in which p is requested, in increasing order ▷ put ∞ at the end of array
    $pointer[p] \leftarrow 1$
$Q \leftarrow$ empty priority queue
for every $t \leftarrow 1$ to T **do**
    $pointer[\rho_t] \leftarrow pointer[\rho_t] + 1$
    if $\rho_t \in Q$ **then**
        Q.increase-key$(\rho_t, times[\rho_t, pointer[\rho_t]])$, **print** "hit", **continue**
    if $Q.size() < k$ **then**
        print "load ρ_t to an empty page"
    else
        $p \leftarrow Q$.extract-max(), **print** "evict p and load ρ_t"
    Q.insert$(\rho_t, times[\rho_t, pointer[\rho_t]])$ ▷ add ρ_t to Q with key value $times[\rho_t, pointer[\rho_t]]$

Outline

- 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:

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

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	$O(n)$	$O(1)$	$O(n)$

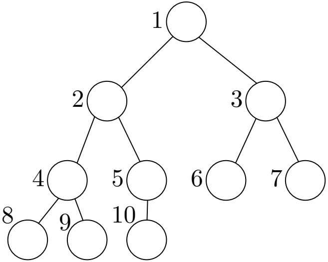
Simple Implementations for Priority Queue

- 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:

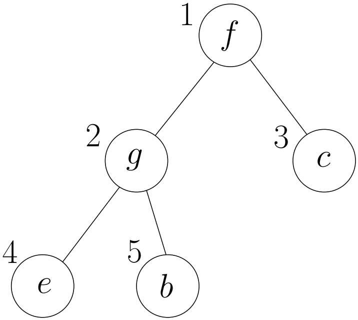


- Nodes are indexed as $\{1, 2, 3, \cdots, s\}$
- Parent of node i: $\lfloor i/2 \rfloor$
- Left child of node i: $2i$
- Right child of node i: $2i + 1$

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

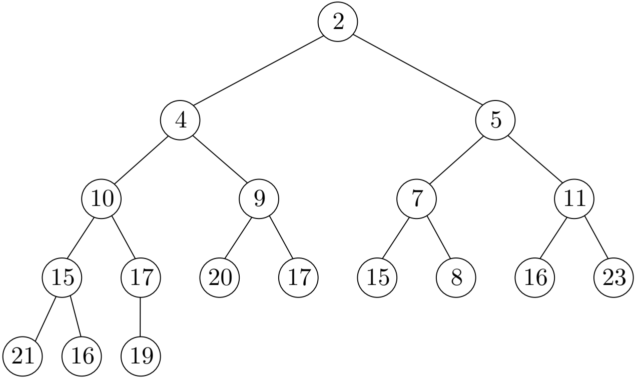


- $s = 5$
- $A = (\text{'}f\text{'}, \text{'}g\text{'}, \text{'}c\text{'}, \text{'}e\text{'}, \text{'}b\text{'})$
- $p[\text{'}f\text{'}] = 1, p[\text{'}g\text{'}] = 2, p[\text{'}c\text{'}] = 3,$
  $p[\text{'}e\text{'}] = 4, p[\text{'}b\text{'}] = 5$

Heap

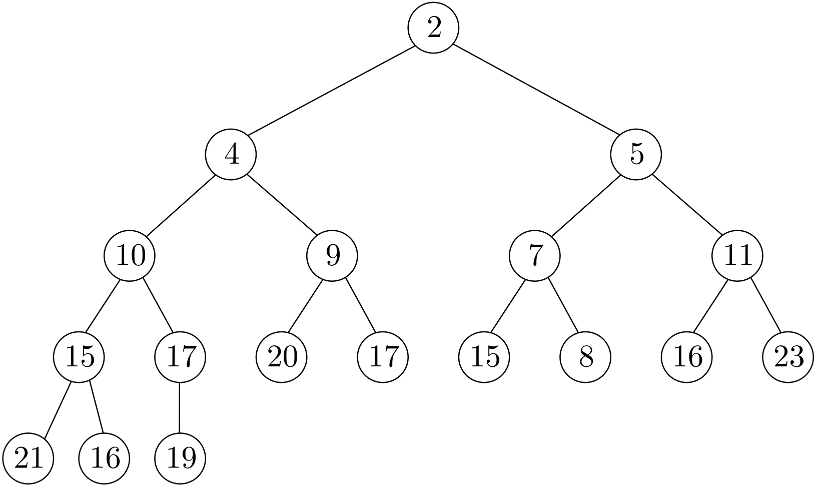
The following heap property is satisfied:

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

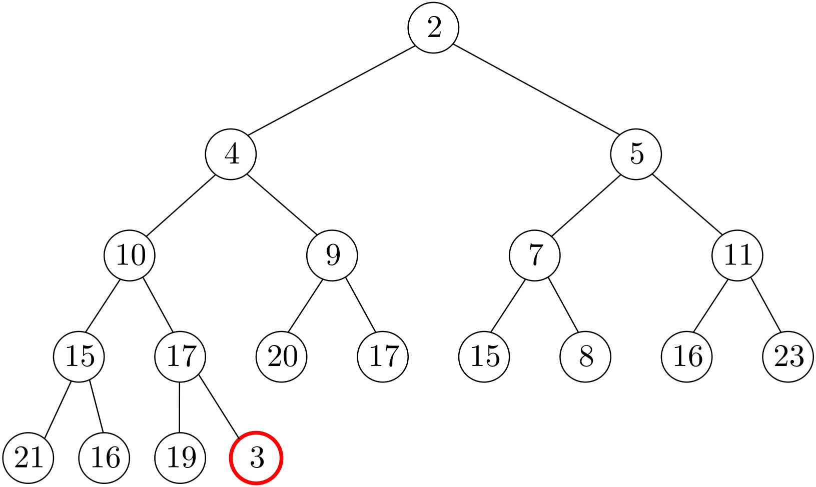


A heap. Numbers in the circles denote key values of elements.

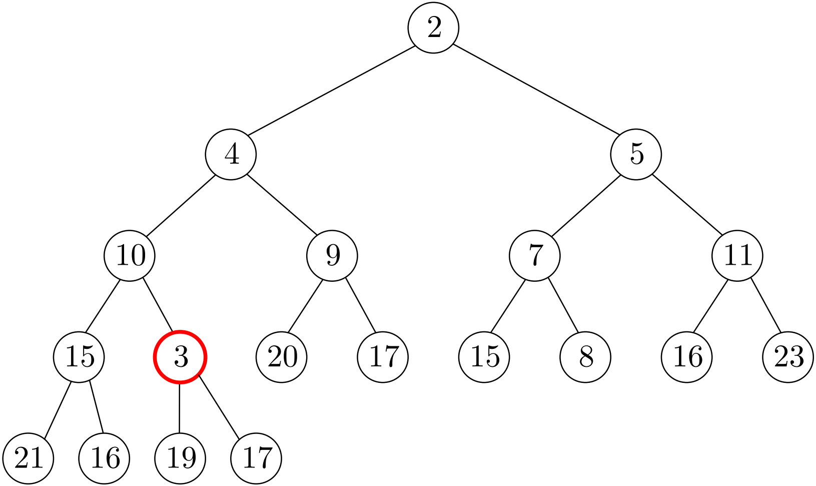
insert(v, key_value)



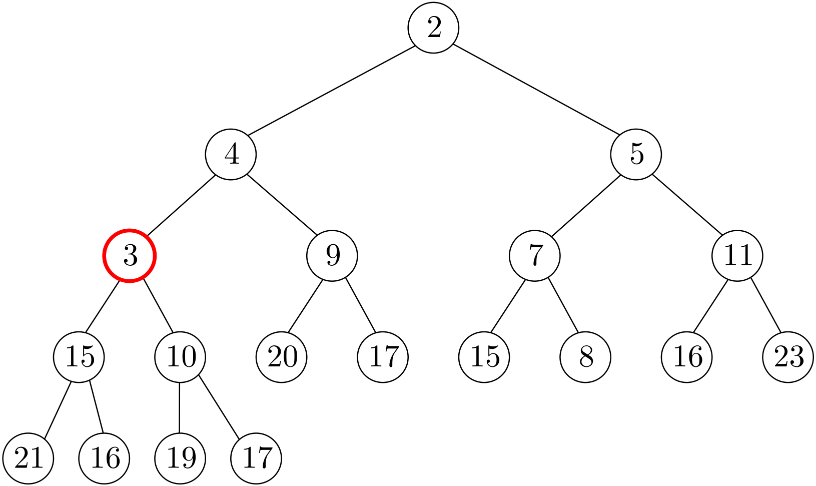
insert(v, key_value)



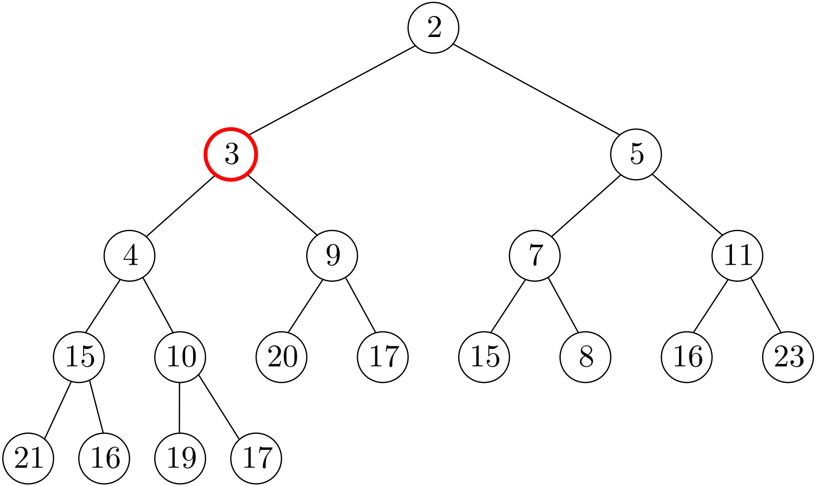
insert(v, key_value)



insert(v, key_value)



insert(v, key_value)



insert(v, key_value)

$s \leftarrow s + 1$
$A[s] \leftarrow v$
$p[v] \leftarrow s$
$key[v] \leftarrow key\_value$
heapify_up(s)

heapify-up(i)

while $i > 1$ **do**
    $j \leftarrow \lfloor i/2 \rfloor$
    if $key[A[i]] < key[A[j]]$ **then**
        swap $A[i]$ and $A[j]$
        $p[A[i]] \leftarrow i$, $p[A[j]] \leftarrow j$
        $i \leftarrow j$
    else break