- Problem 3

  - What are your observations about the query plans for the scanning and sorting of such differently sized bags S?
    - Below is time to scan the entire data, the scan time grows in relation to size of S i.e O(N). The planning time seems constant.

SCAN

Data Output | Explain | Messages | Notifications | Scratch Pad

	size of relation S integer	Avg execution time to scan S (in ms): SELECT x FROM S numeric
1	10	0.007
2	100	0.012
3	1000	0.068
4	10000	0.658
5	100000	6.579
6	1000000	65.986
7	10000000	737.383
8	100000000	7773.957

SORT

	size of relation S integer	Avg execution time to sort S (in ms): SELECT x FROM S order by numeric
1	10	0.019
2	100	0.032
3	1000	0.194
4	10000	2.490
5	100000	29.459
6	1000000	343.130
7	10000000	3854.534
8	100000000	30024.235

  - What do you observe about the execution time to sort S as a function of n?
    - For relatively small data and based on `work_mem` parameter, the quick sort is used to sort data but as soon as the data size grows instead of quick sort the postgres uses external merge sort, with time complexity of $O(n \log_B(n))$ where B is block size
  - Does this conform with the formal time complexity of (external) sorting? Explain.
    - Since external sort works in time complexity of $O(n \log_B(n))$ where B is block size. Here we take example of block size as 64kb = 64000 bytes thus to sort 100000 records we will take O(100000 log64000 (100000)) * time for I/O operation we get corresponding time. All the above experiments are done with work_mem setting of 1.5 GB

  - It is possible to set the working memory of PostgreSQL using the set work mem command. For example, set work mem = '16MB' sets the working memory to 16MB.8 The smallest possible working memory in postgreSQL is 64kB and the largest depends on your computer memory size. But you can try for example 1GB. Repeat question 3a for memory sizes 64kB and 1GB and report your observations.
  -

64 KB Scan

	size of relation S integer	Avg execution time to scan S (in ms): SELECT x FROM S numeric
1	10	0.026
2	100	0.012
3	1000	0.070
4	10000	0.647
5	100000	6.599
6	1000000	66.202
7	10000000	744.759
8	100000000	7869.021

64 KB Sort

Data Output | Explain | Messages | Notifications | Scratch Pad

	size of relation S integer	Avg execution time to sort S (in ms): SELECT x FROM S order by numeric
1	10	0.015
2	100	0.028
3	1000	0.620
4	10000	3.353
5	100000	39.664
6	1000000	309.161
7	10000000	3131.792
8	100000000	47874.703

1 Gb Scan

1 Gb Sort

Data Output | Explain | Messages | Notifications | Scratch Pad

	size of relation S integer	Avg execution time to scan S (in ms): SELECT x FROM S numeric
1	10	0.007
2	100	0.012
3	1000	0.068
4	10000	0.675
5	100000	6.629
6	1000000	66.294
7	10000000	745.419
8	100000000	7896.808

	size of relation S integer	Avg execution time to sort S (in ms): SELECT x FROM S order by x numeric
1	10	0.016
2	100	0.027
3	1000	0.199
4	10000	2.642
5	100000	29.376
6	1000000	345.323
7	10000000	3851.359
8	100000000	40658.351

Since the work_mem parameter sets the block size with which the postgres can work, setting the large block size certainly helps the sorting when data size grows since postgres can accommodate more data in memory but for smaller sizes of data it hardly makes any difference. In case of sorting data since the external merge sort has log N to base B in time complexity calculation the large size of B reduces the total time for execution.

- I was unable to gather data for sizes above 10^7 for index queries since my system kept crashing. So below is data for index queries.

Create Index

Data Output | Explain | Messages | Notifications | Scratch Pad

	size n of relation S integer	avg execution time to create index indexedS numeric
1	10	0.802
2	100	1.936
3	1000	14.114
4	10000	143.435
5	100000	1035.816
6	1000000	6969.548
7	10000000	51799.617

Sort Index

Data Output | Explain | Messages | Notifications | Scratch Pad

	size n of relation S integer	avg execution time to create index indexedS numeric
1	10	7329.073
2	100	7275.081
3	1000	7247.048
4	10000	7289.490
5	100000	7292.106
6	1000000	7262.839
7	10000000	7376.858

- The B+ Tree creation takes time of order O(N) thus we can see exponential growth in index creation.
- The B+ Tree has advantage in terms of range comparison, however with each range search after reading the respective pointers to record we must fetch actual data as well thus we are able to see some overhead in term of range search.

- Problem 7
  - block size = 4096 bytes
  - block-address size = 9 bytes
  - block access time (I/O operation) = 10 ms (micro seconds)
  - record size = 150 bytes
  - record primary key size = 8 bytes

Here if n is number of keys a particular node in B+ tree can hold then we can have

$$8*n + 9*(n+1) \leq 4096$$

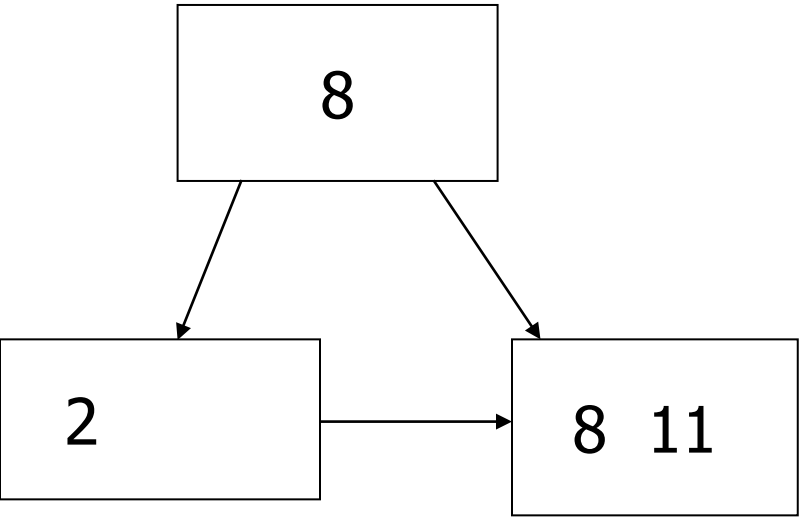
Thus, we can have n=240 keys in any B+ tree node.

1. To find any record we will require $\log_n N$ (+ 1 IO operation to read the actual record)

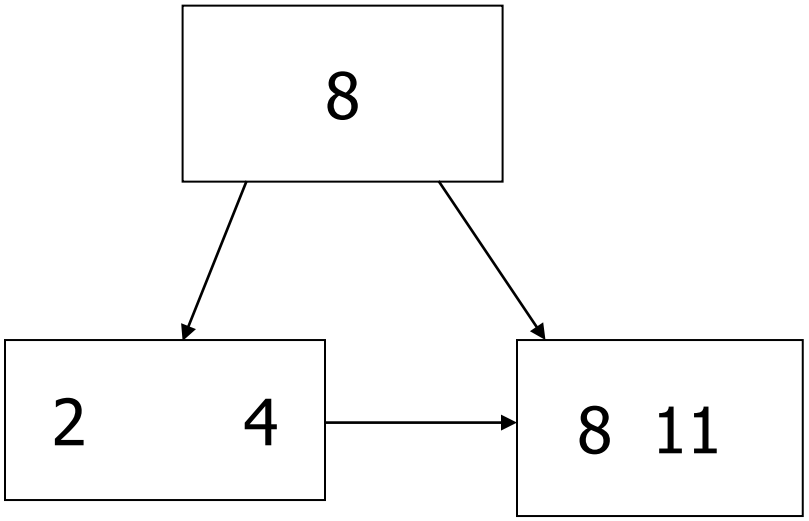
   Here the total number of keys is n = 240 and total number of records can be 10^9

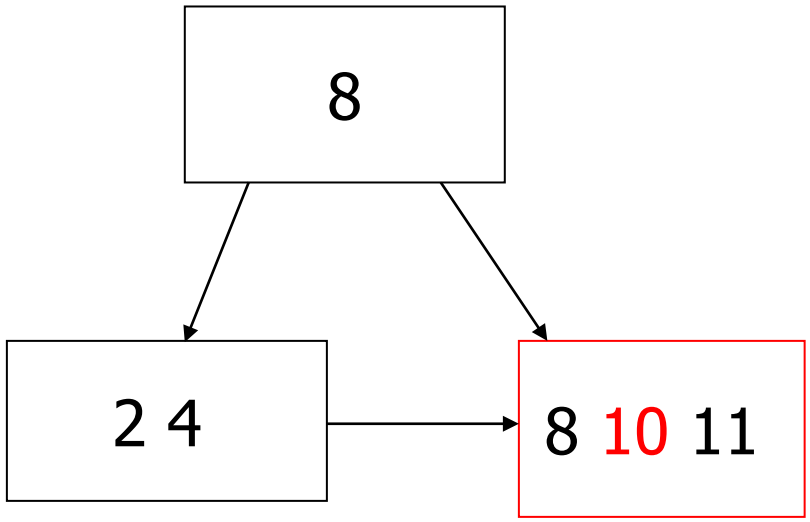
   Thus, total time to find the key will be $\log_{240} 10^9 = 9 * \log_{240} 10 = 3.781179 * block\ access\ time = 3.781179 * 10 = 37.81$ ms

2. To insert any record in B+ Tree it will take $\log_n N = \log_{240} 10^9 = 9 * \log_{240} 10 = 3.781179 * block\ access\ time = 3.781179 * 10 = 37.81$ ms
3. Memory requirements
   a. The first level
      i. The root node will have 240 keys with 241 pointers to child node thus to store root node the memory require will be 4096 bytes
   b. The second level:
      i. Since each pointer in root node points to another node with 240 keys and 241 pointers the total memory will be 241*4096 bytes
      ii. Thus, to store level 1 and level 2 we will require: 4096 + 241 * 4096 = 4096*242 bytes = 0.9453125 MB
   c. The third level
      i. At second level we have total of 241 nodes with each node having another 241 blocks pointing to next level thus we have total of
         1. 241*241 blocks at third level
         2. Thus total memory required will be: 241*241*4096 = 237,899,776 bytes = 226.87890625 MB.
         3. To store all three levels total memory requirements will be: 0.9453125 + 226.87890625 = 227.82 MB



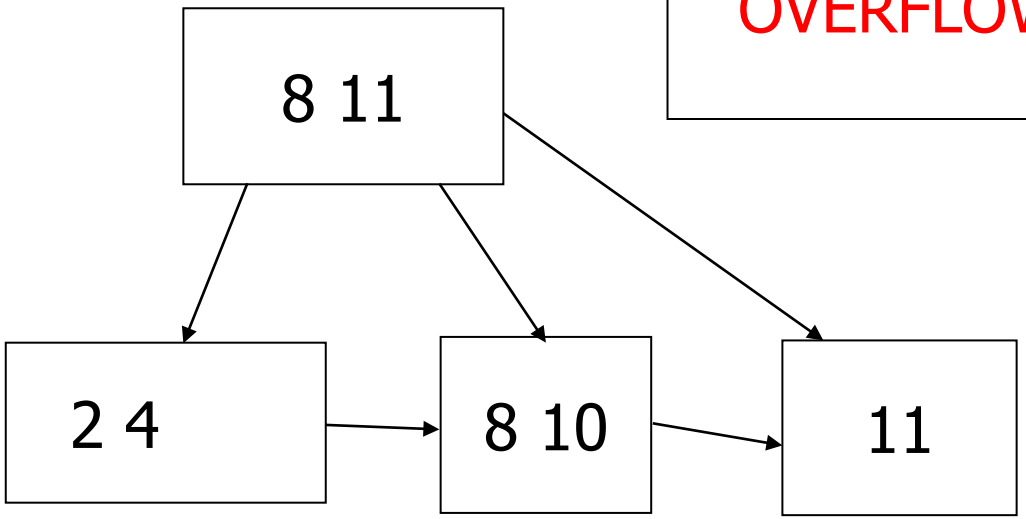
INSERT 4

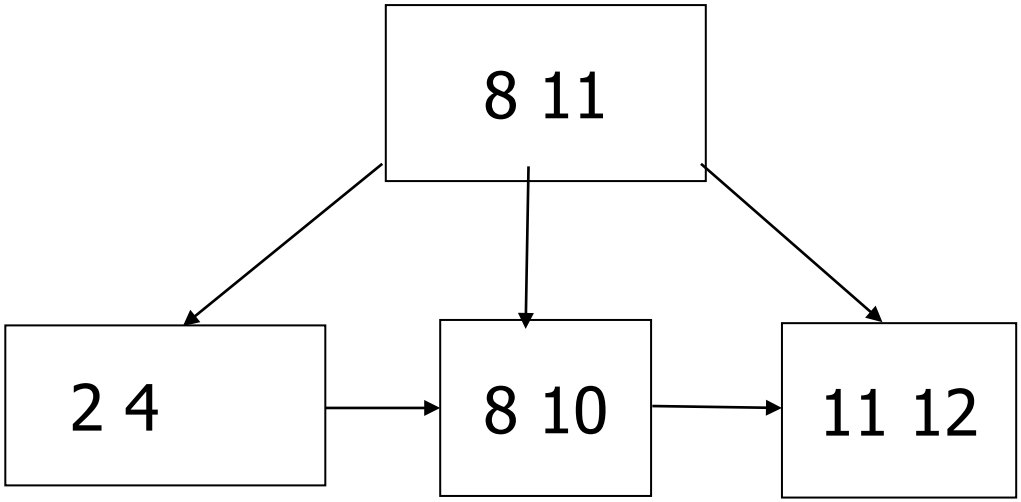




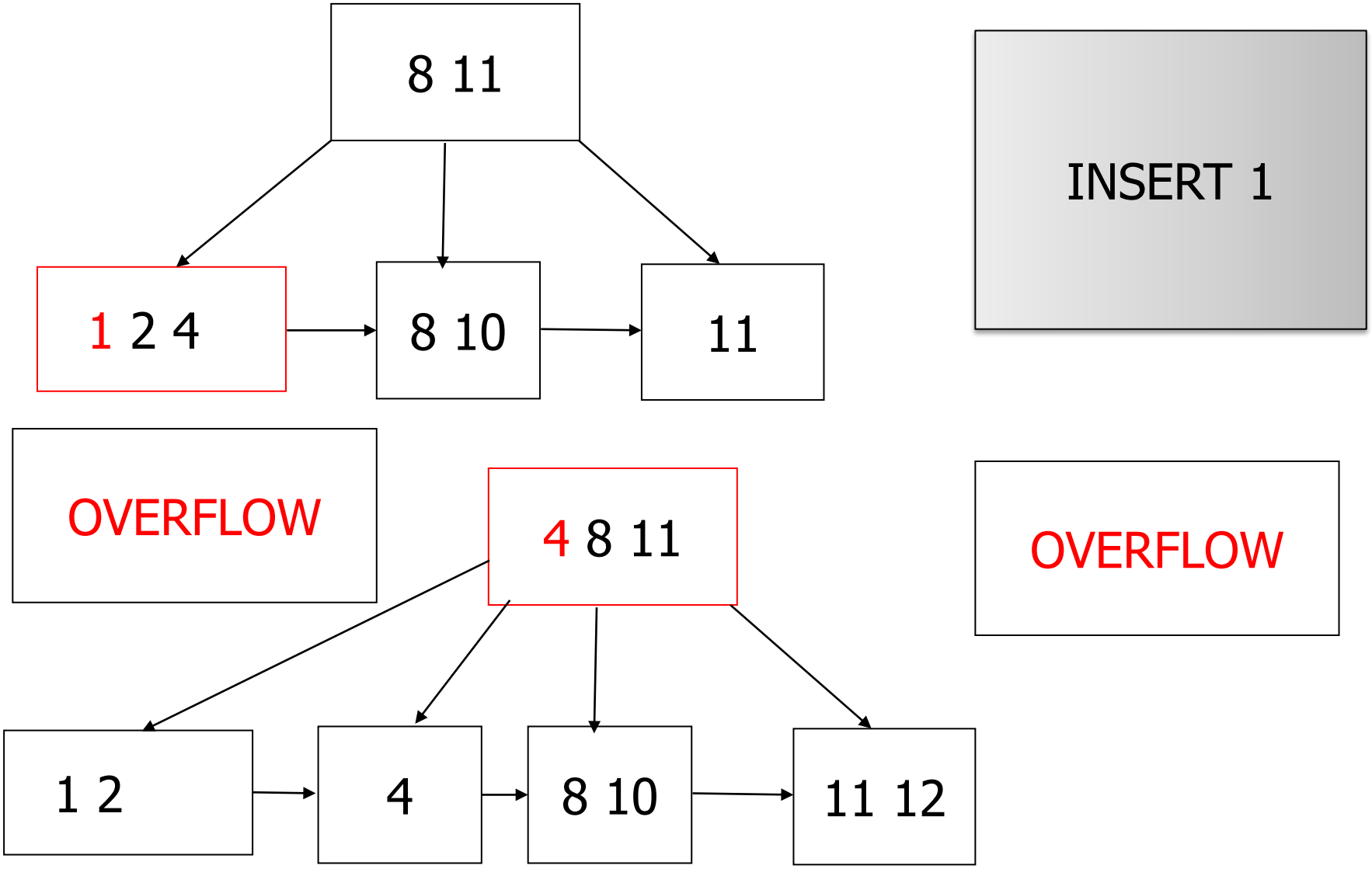
INSERT 10

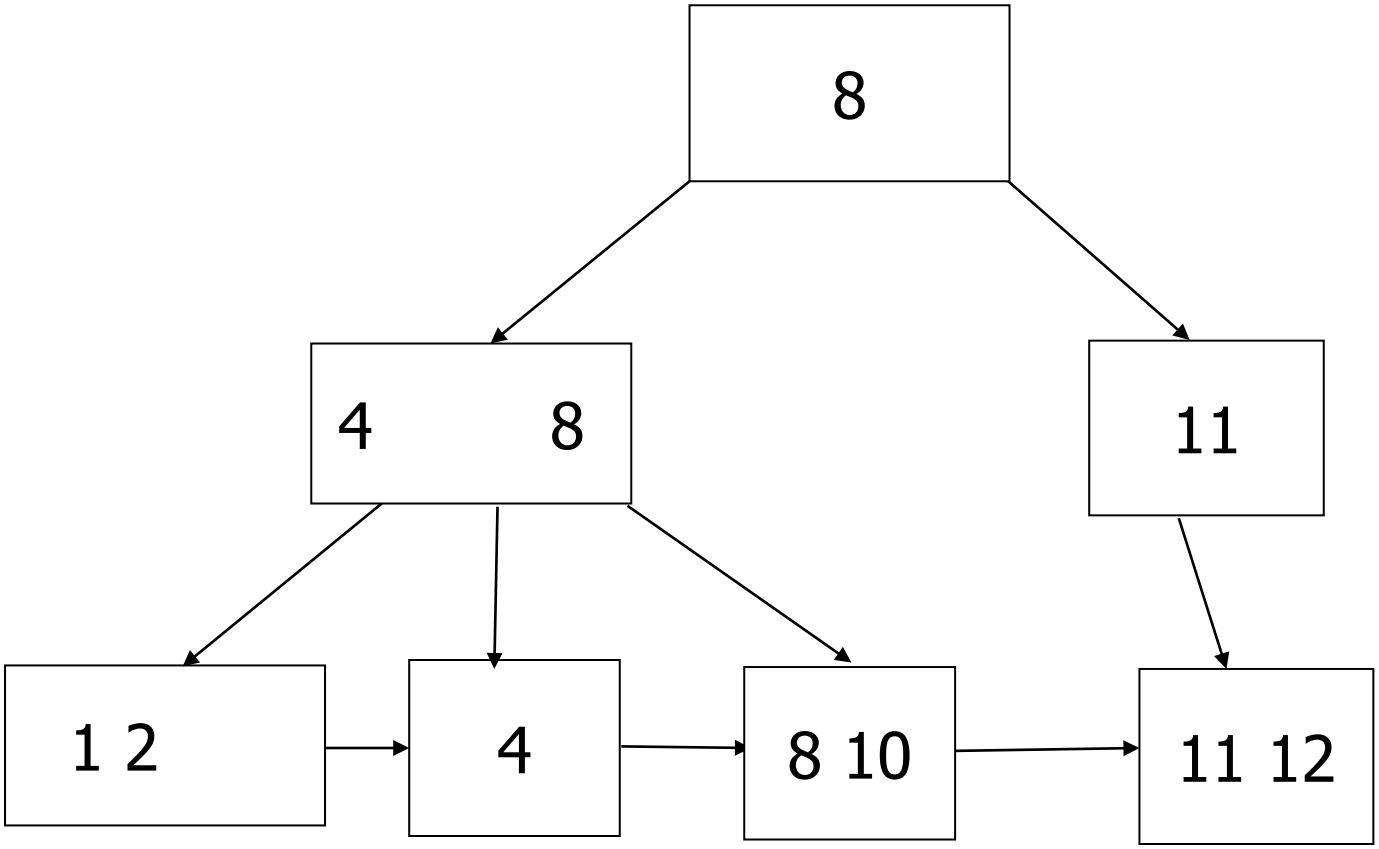
OVERFLOW

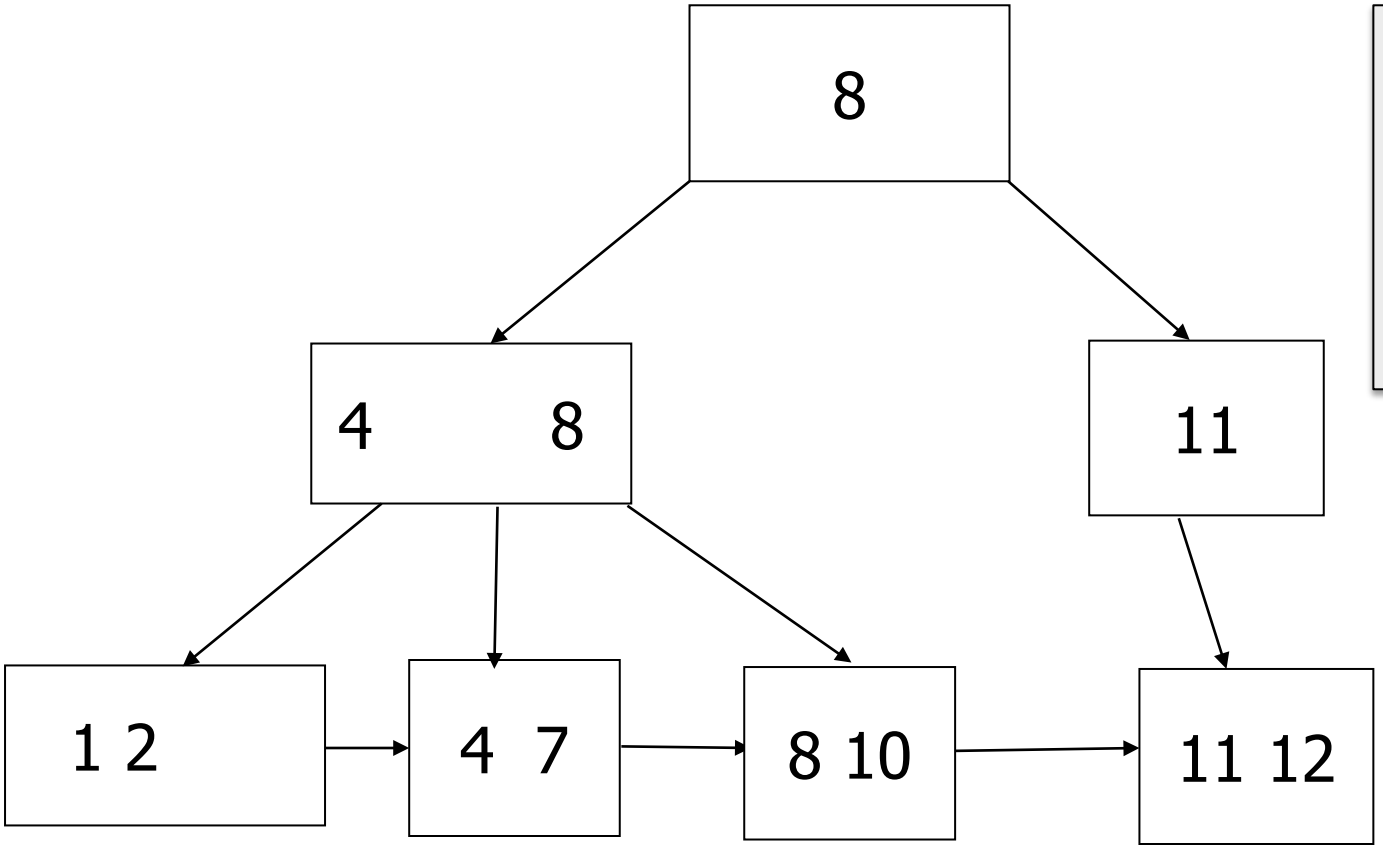




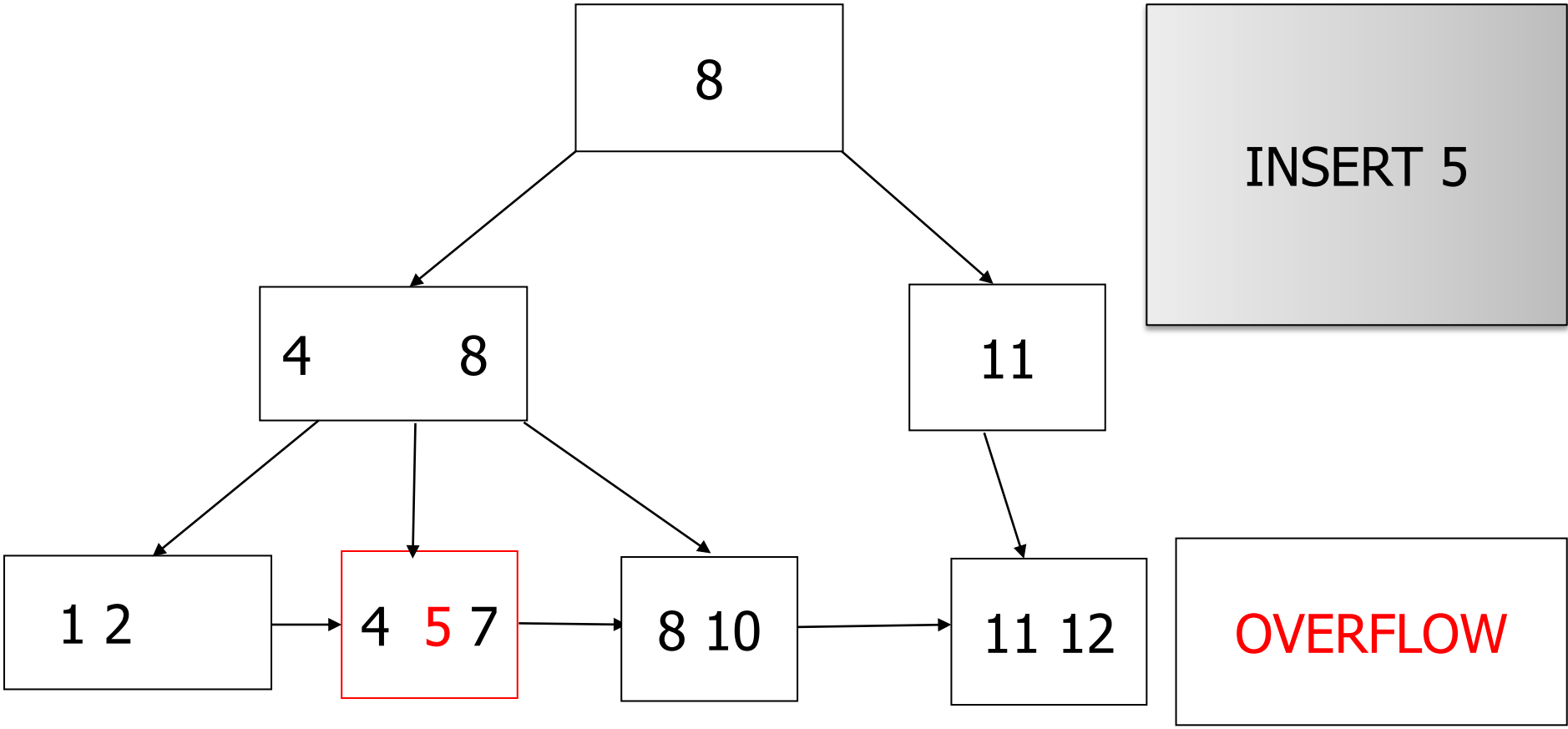
INSERT 12

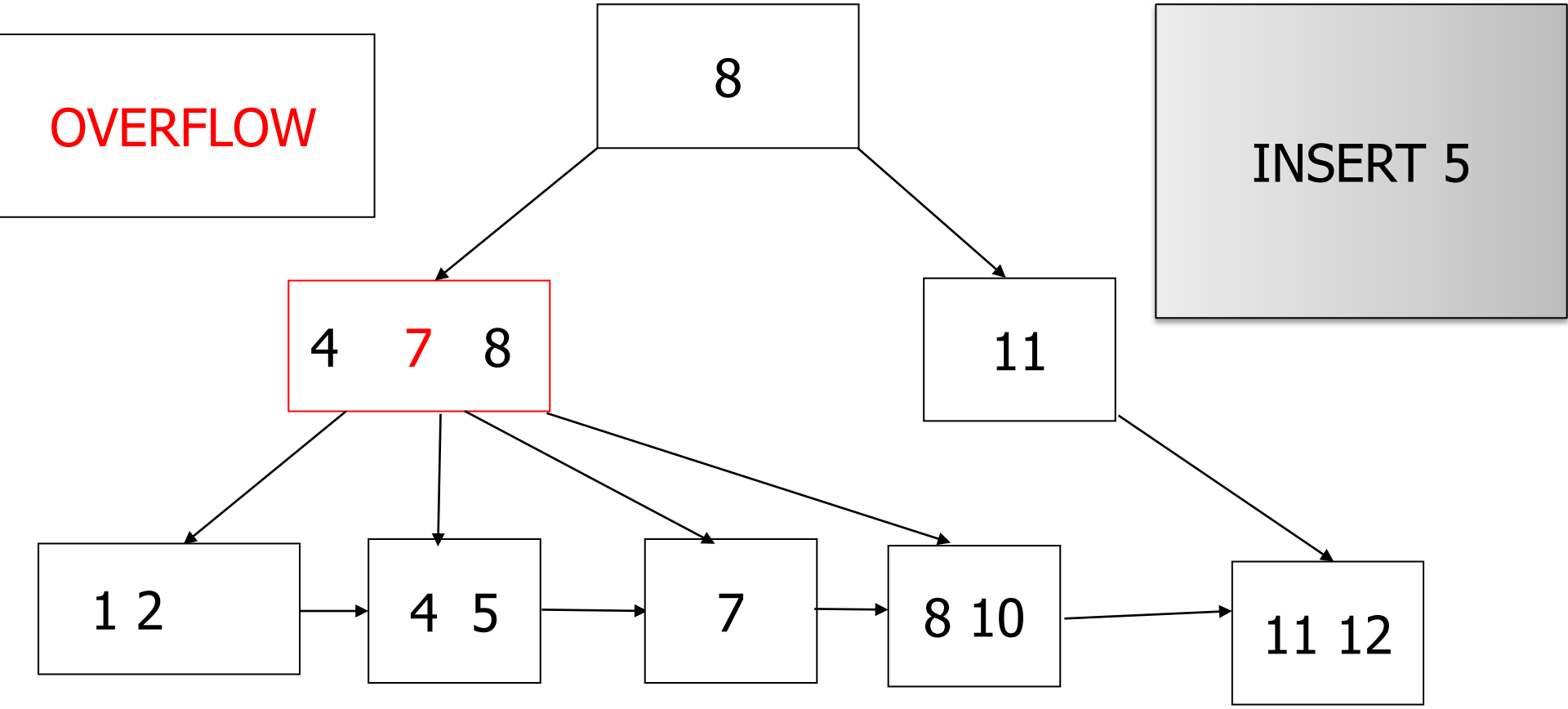

8 11
INSERT 1
1 2 4
8 10
11
OVERFLOW
4 8 11
OVERFLOW
1 2
4
8 10
11 12

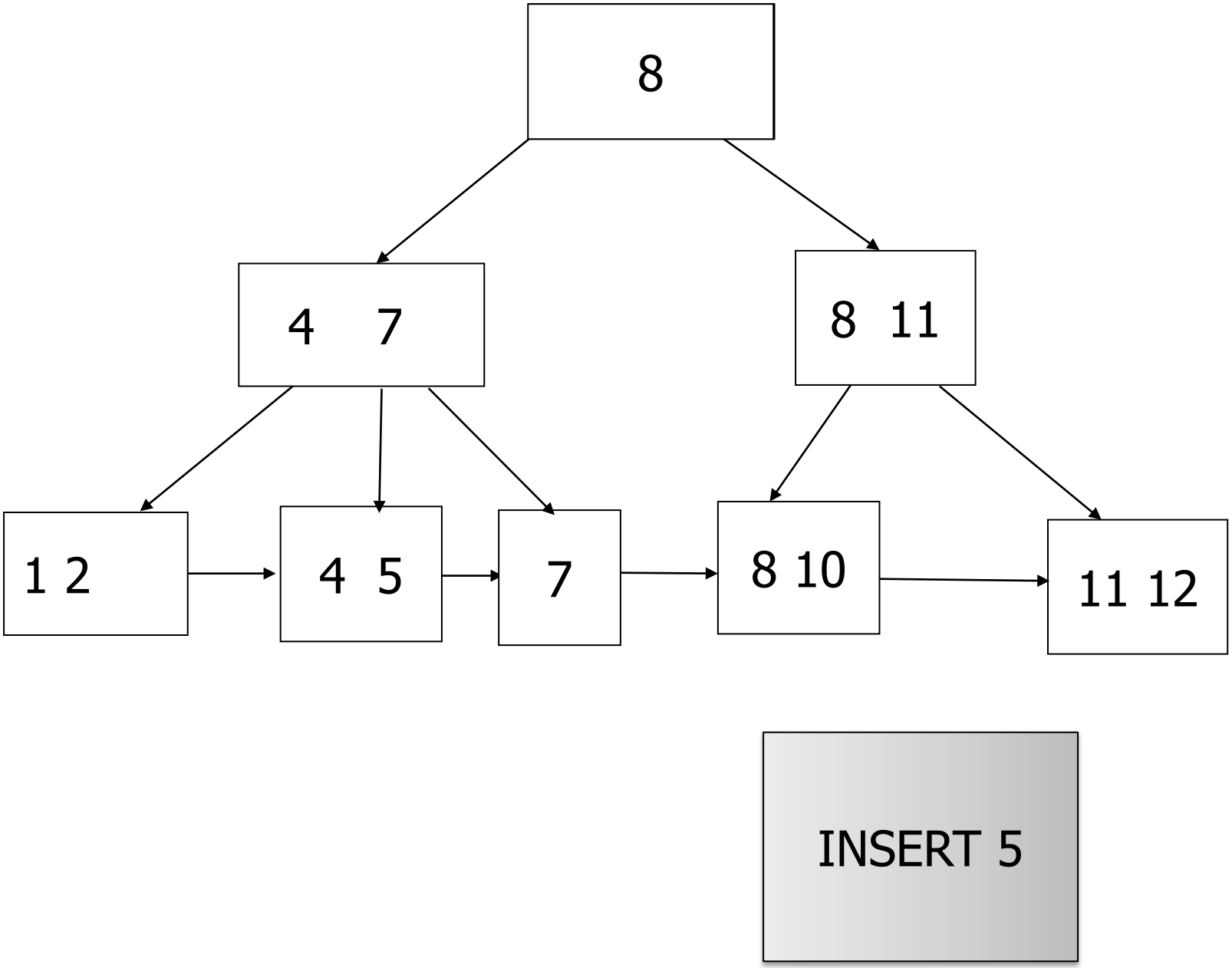

8
4 8
11
1 2
4
8 10
11 12

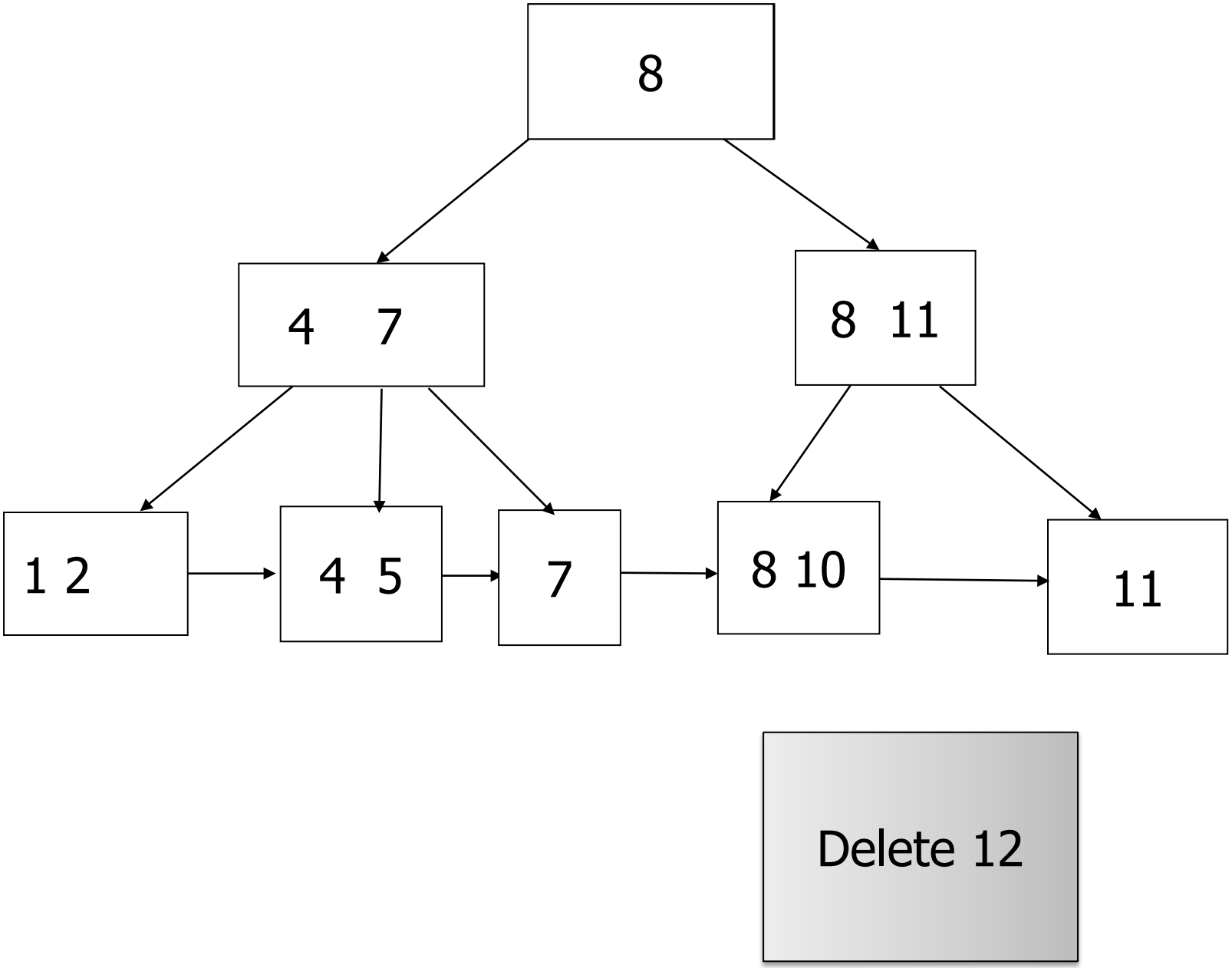


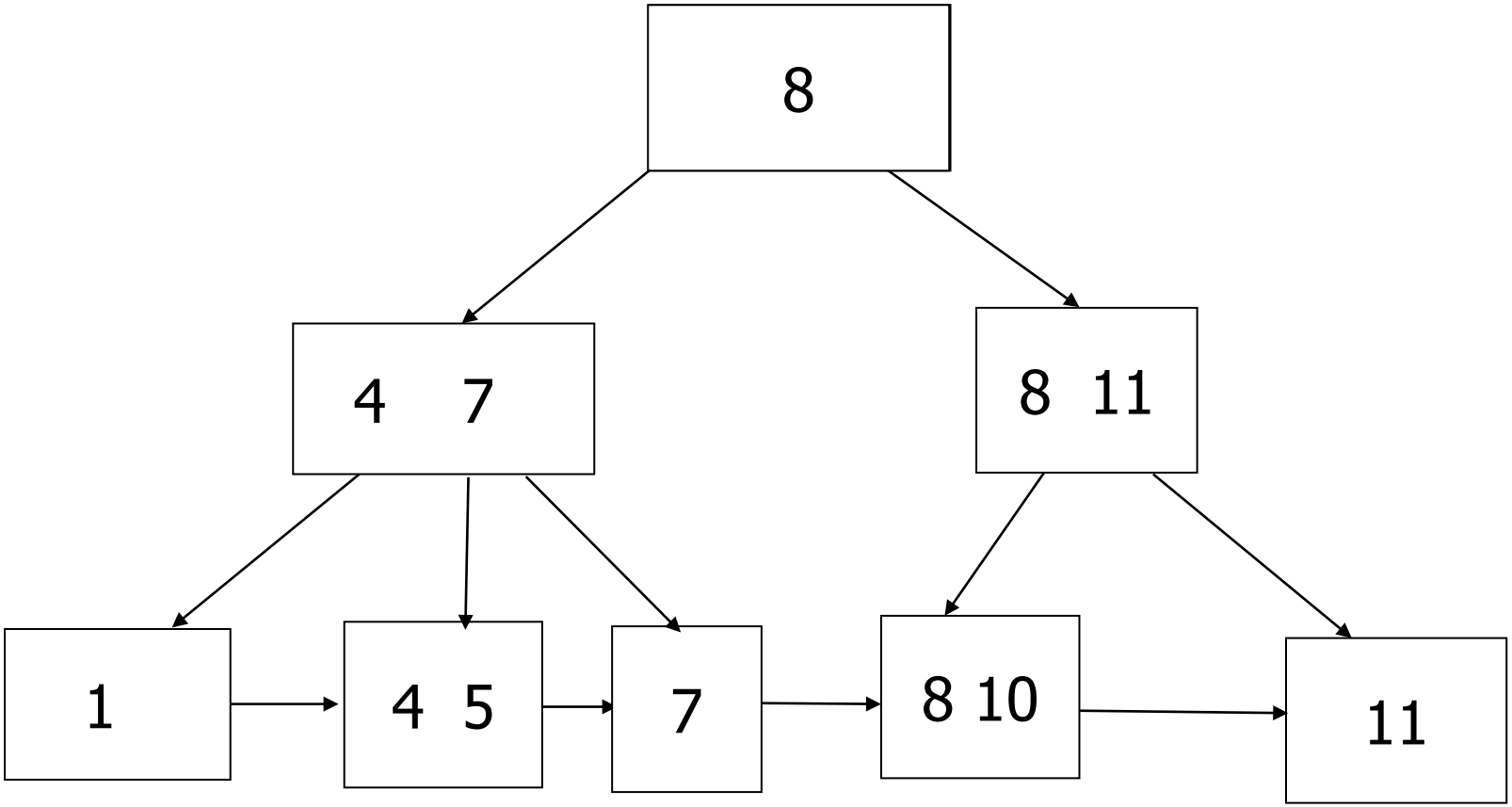
INSERT 7


8
4 8
11
1 2
4 5 7
8 10
11 12
INSERT 5
OVERFLOW

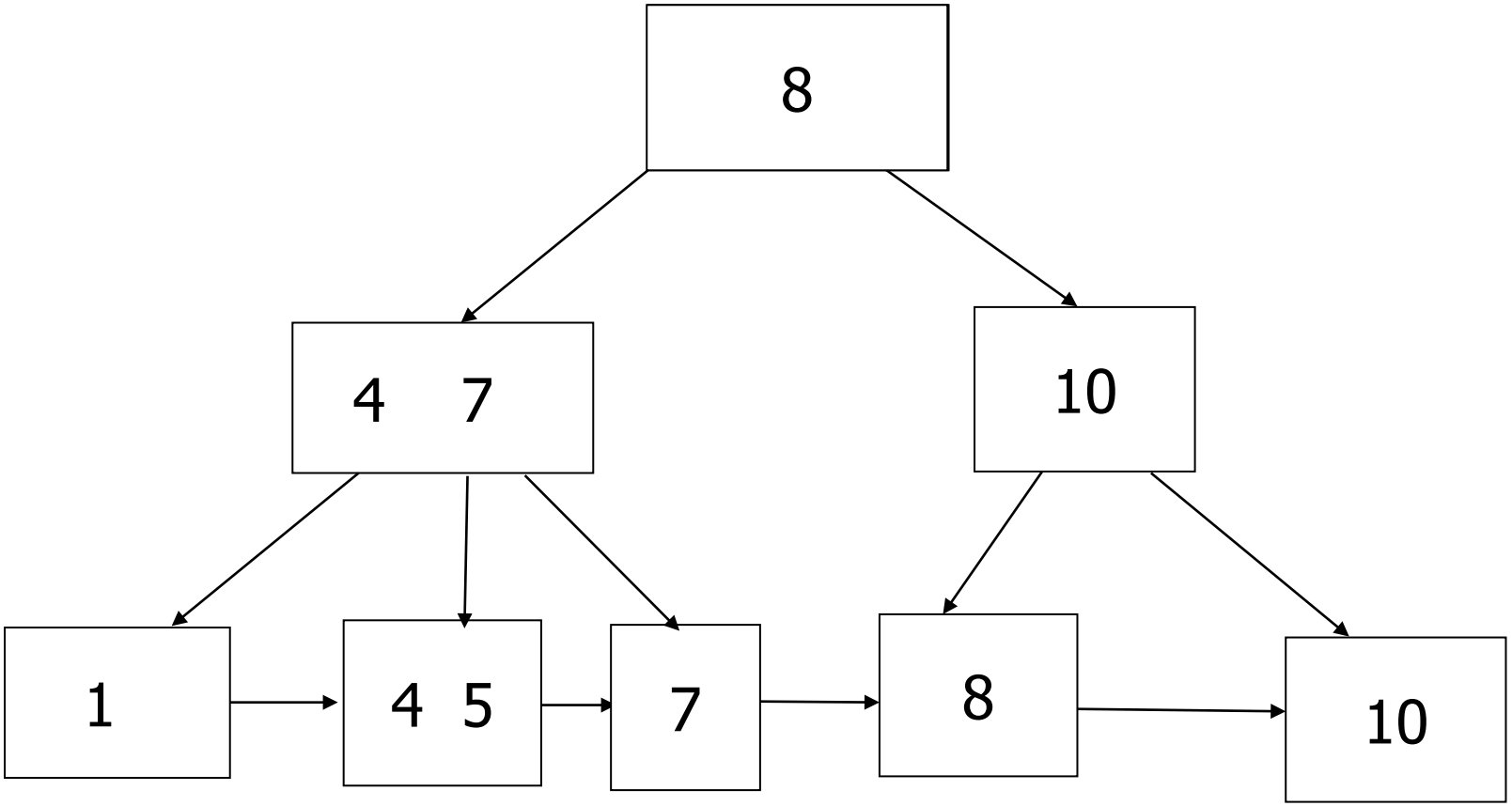

OVERFLOW
8
INSERT 5
4 7 8
11
1 2
4 5
7
8 10
11 12


8
4 7
8 11
1 2
4 5
7
8 10
11 12
INSERT 5

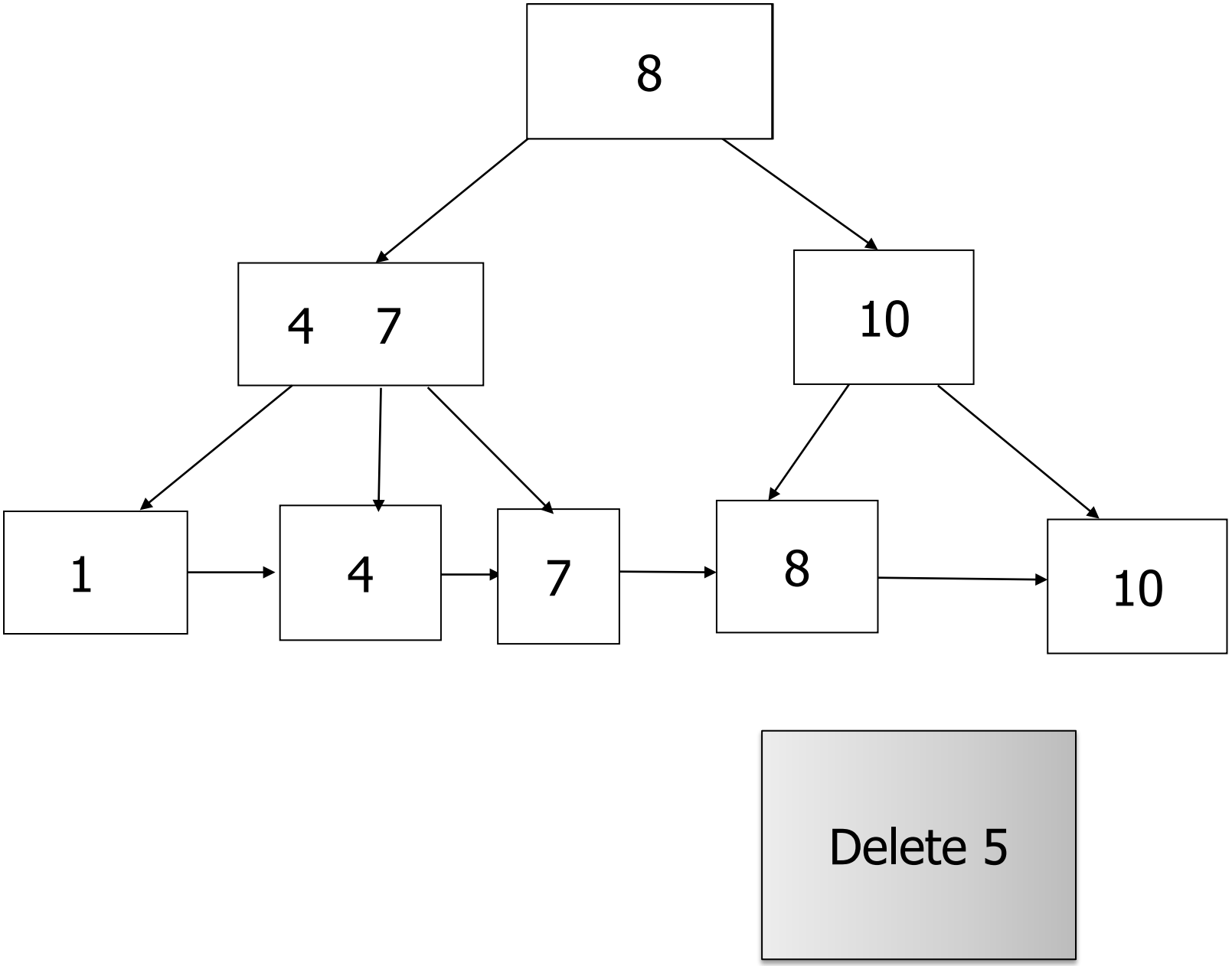

8
4 7
8 11
1 2
4 5
7
8 10
11
Delete 12



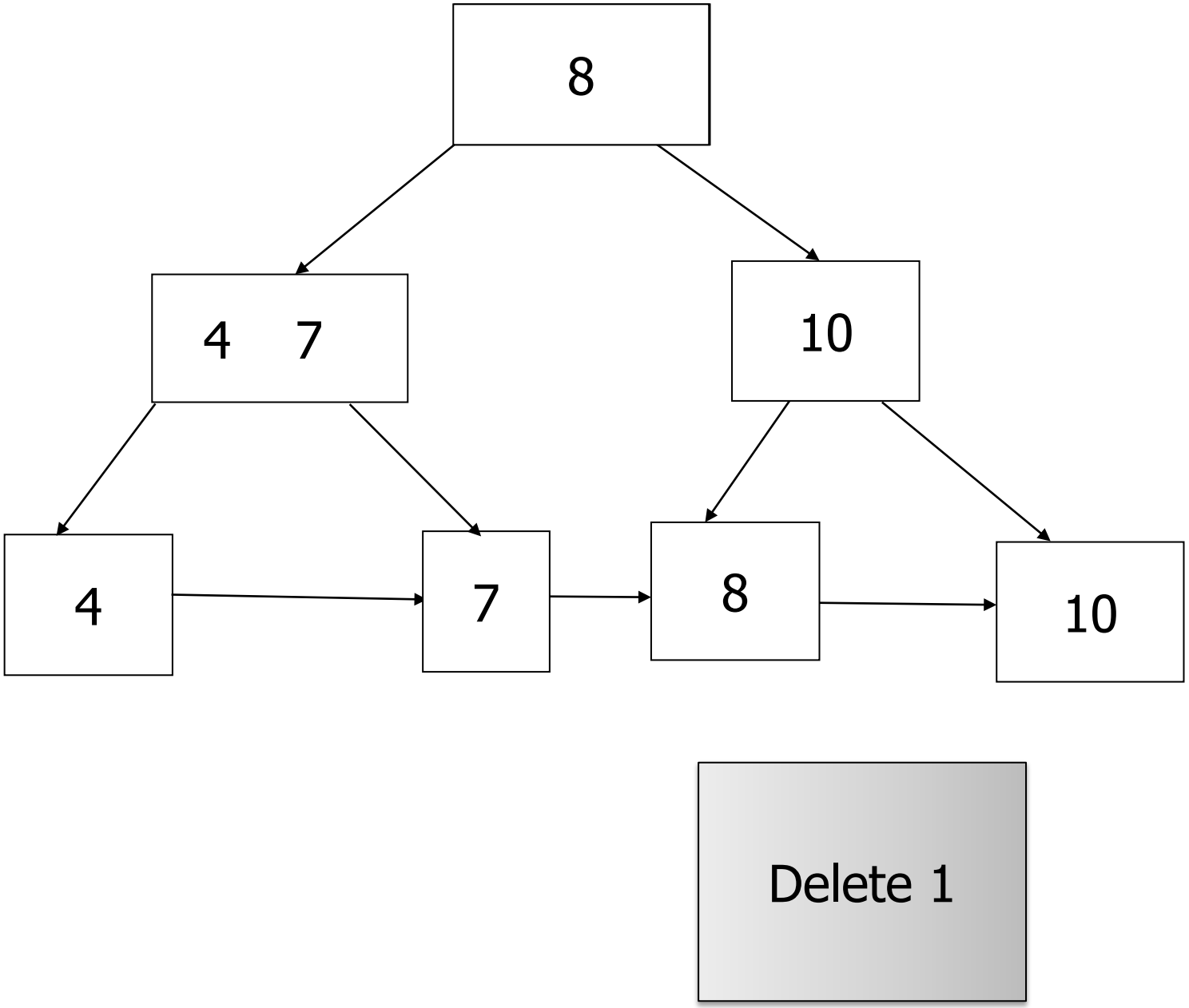
Delete 2

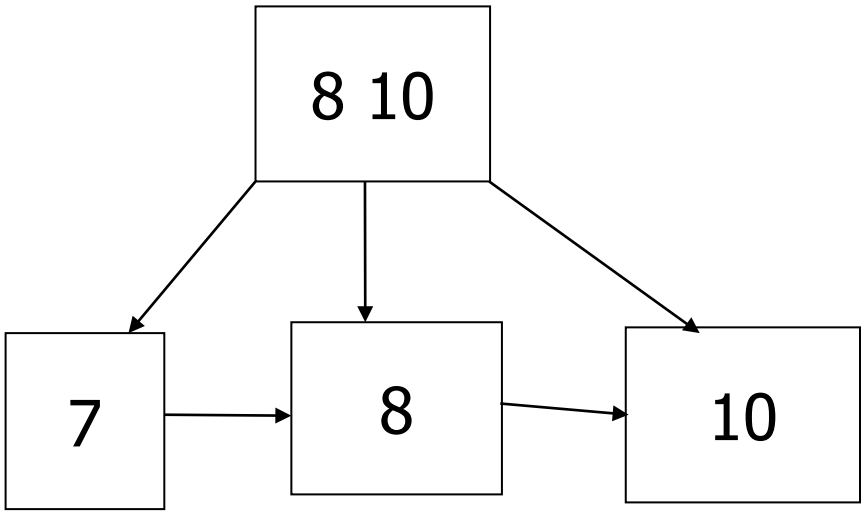


Delete 11

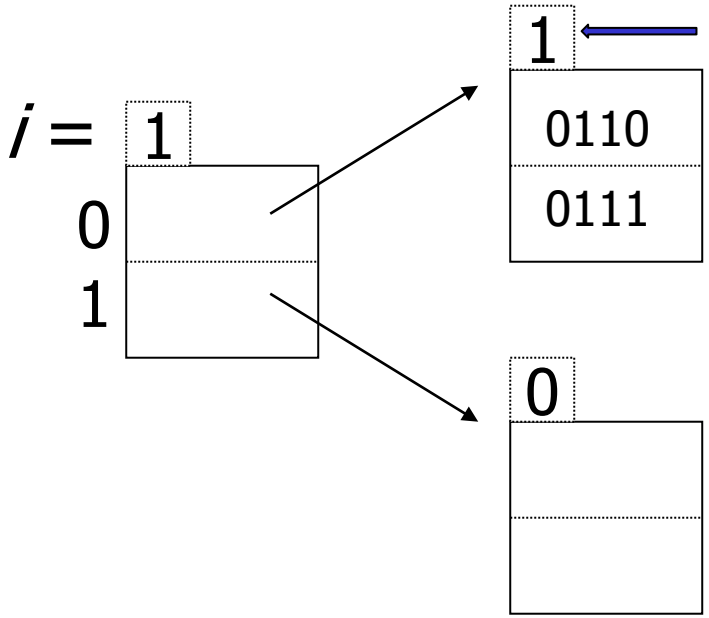


Delete 5


8
4 7
10
4
7
8
10
Delete 1



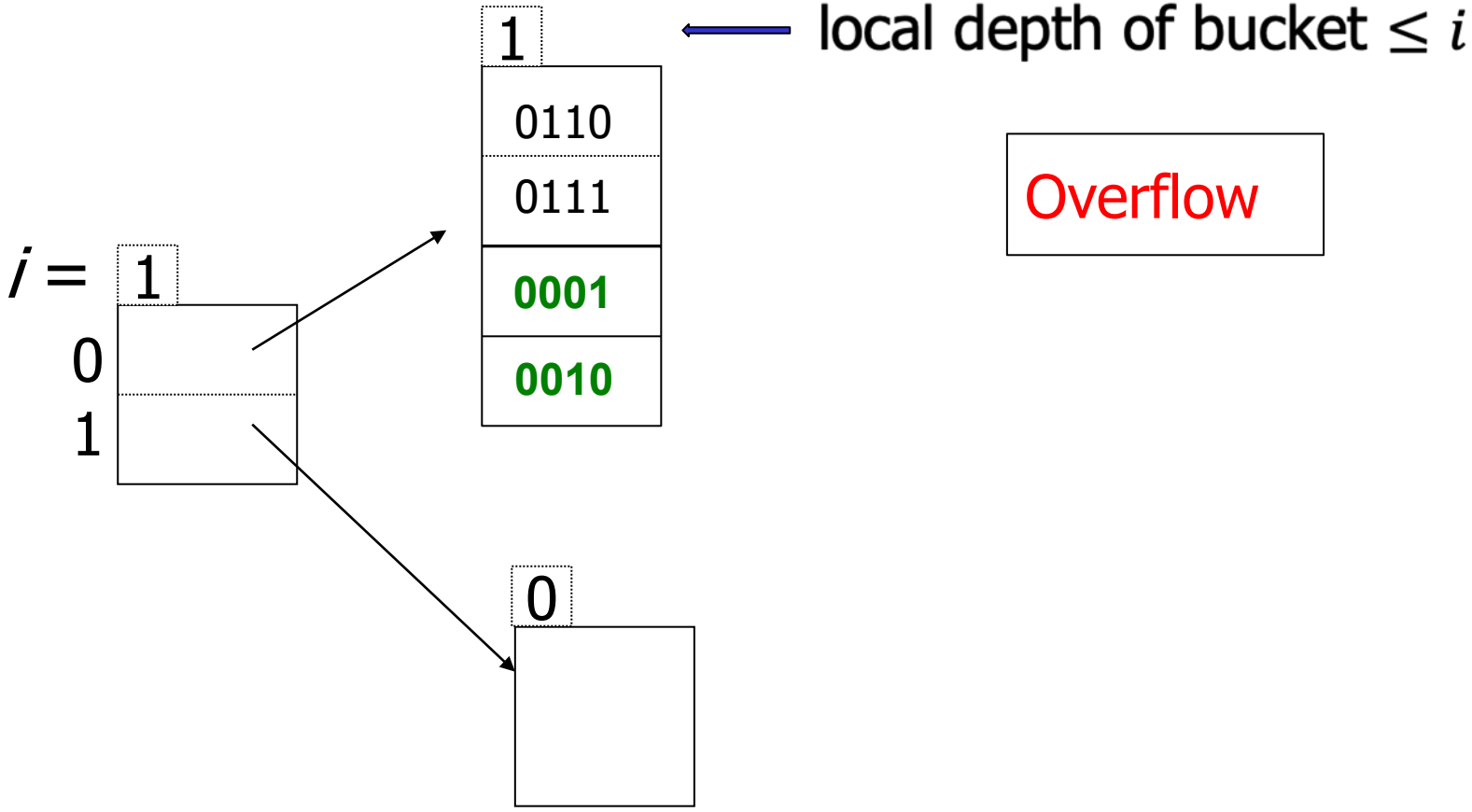
Delete 4



local depth of bucket $\leq i$

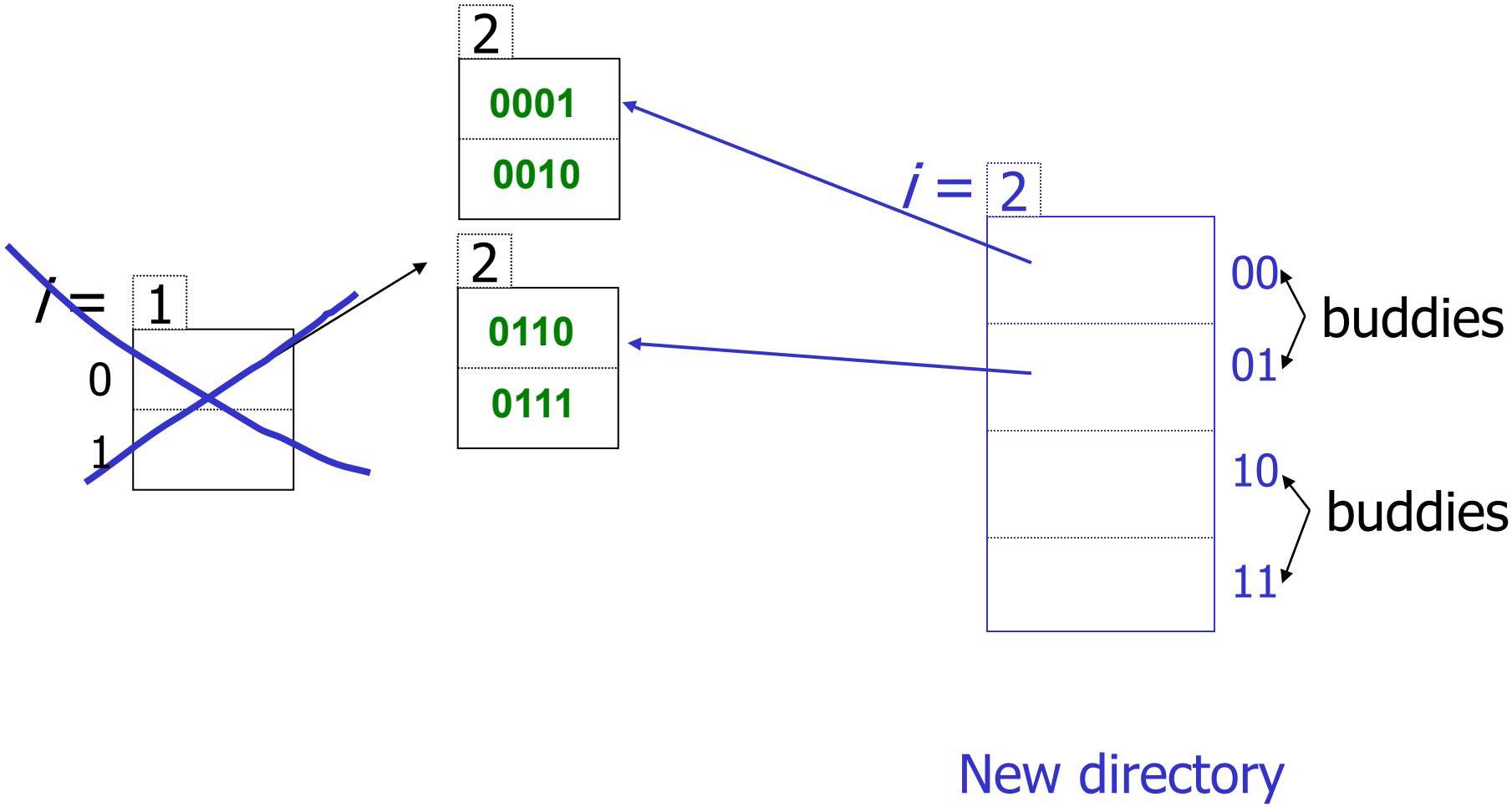
Insert **0110 = 6**

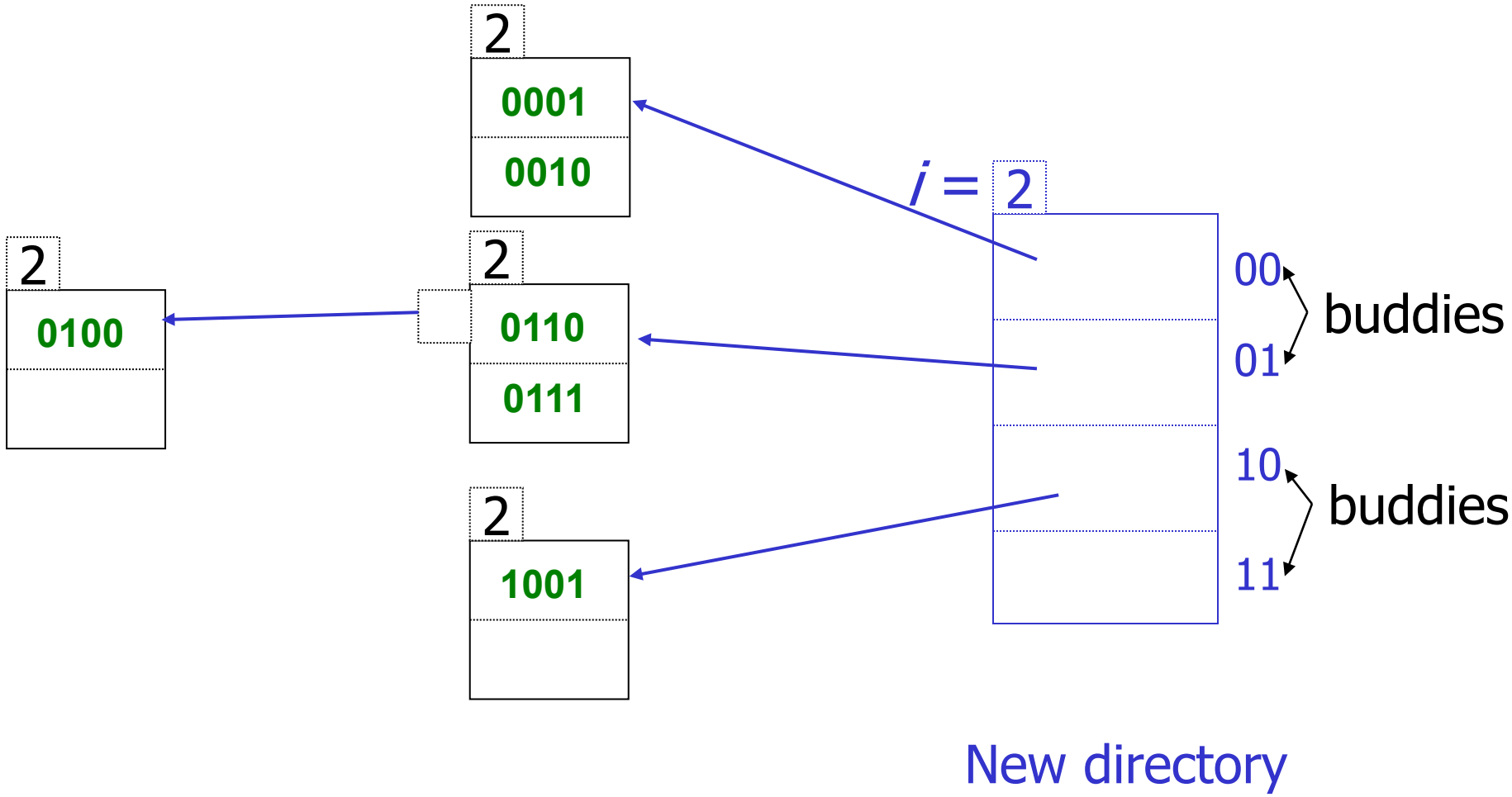
Insert **0111 = 7**



Insert **0001 = 1**

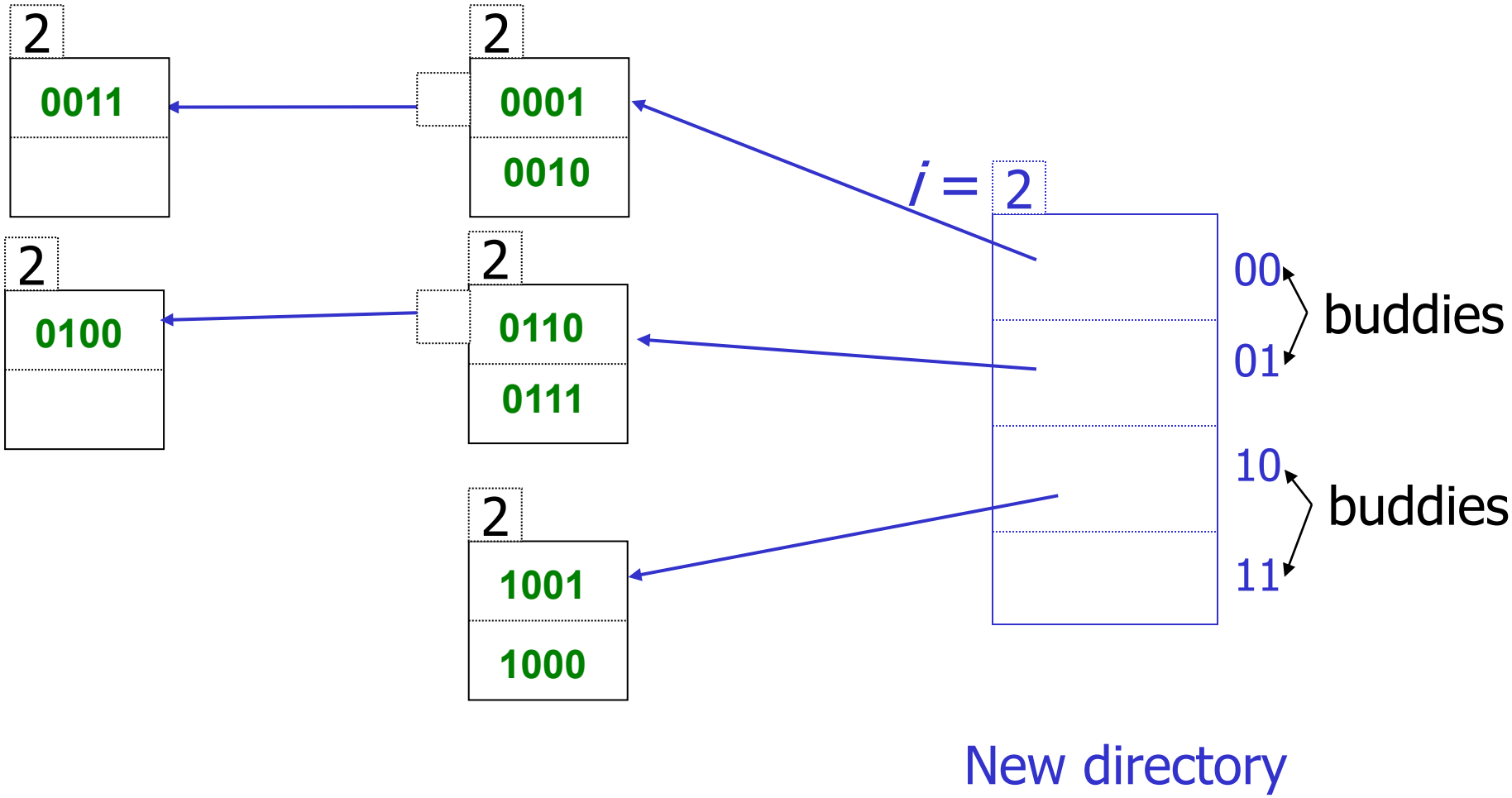
Insert **0010 = 2**


2
0001
0010
2
0110
0111
i = 1
0
1
i = 2
00
01
10
11
buddies
buddies
New directory



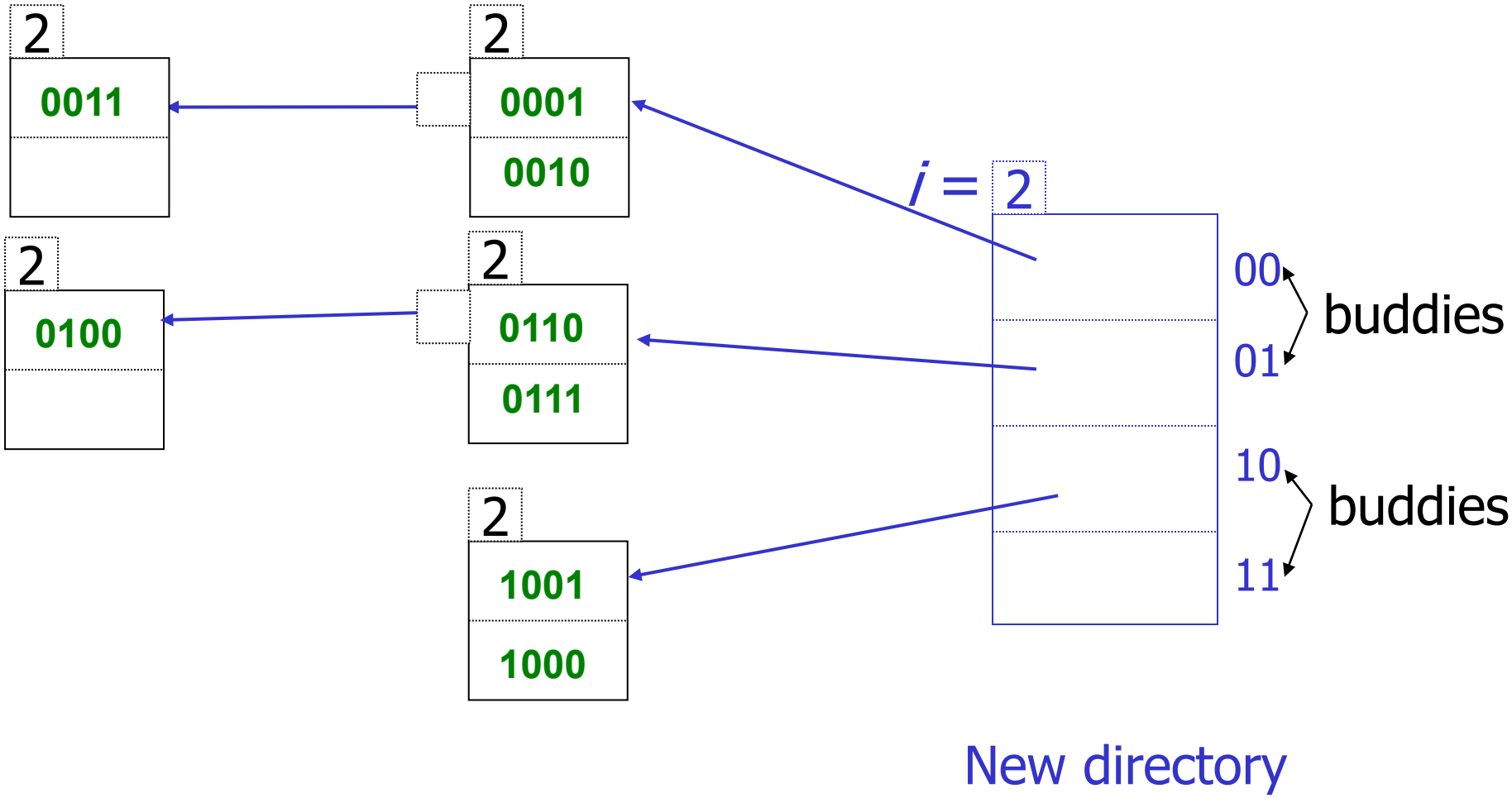
Insert **1001 = 9**

Insert **0100 = 4**



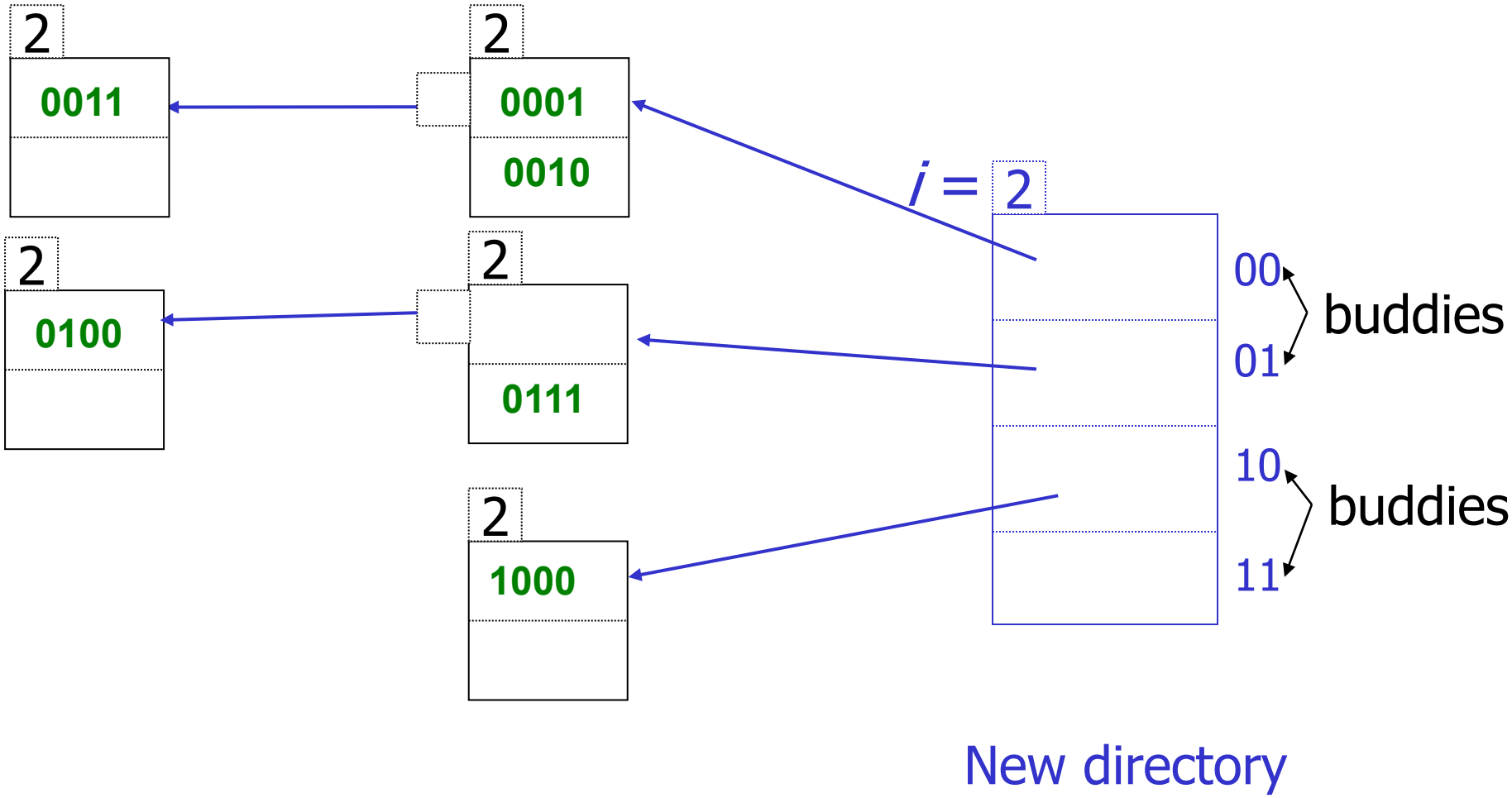
Insert **1000 = 8**

Insert **0011 = 3**



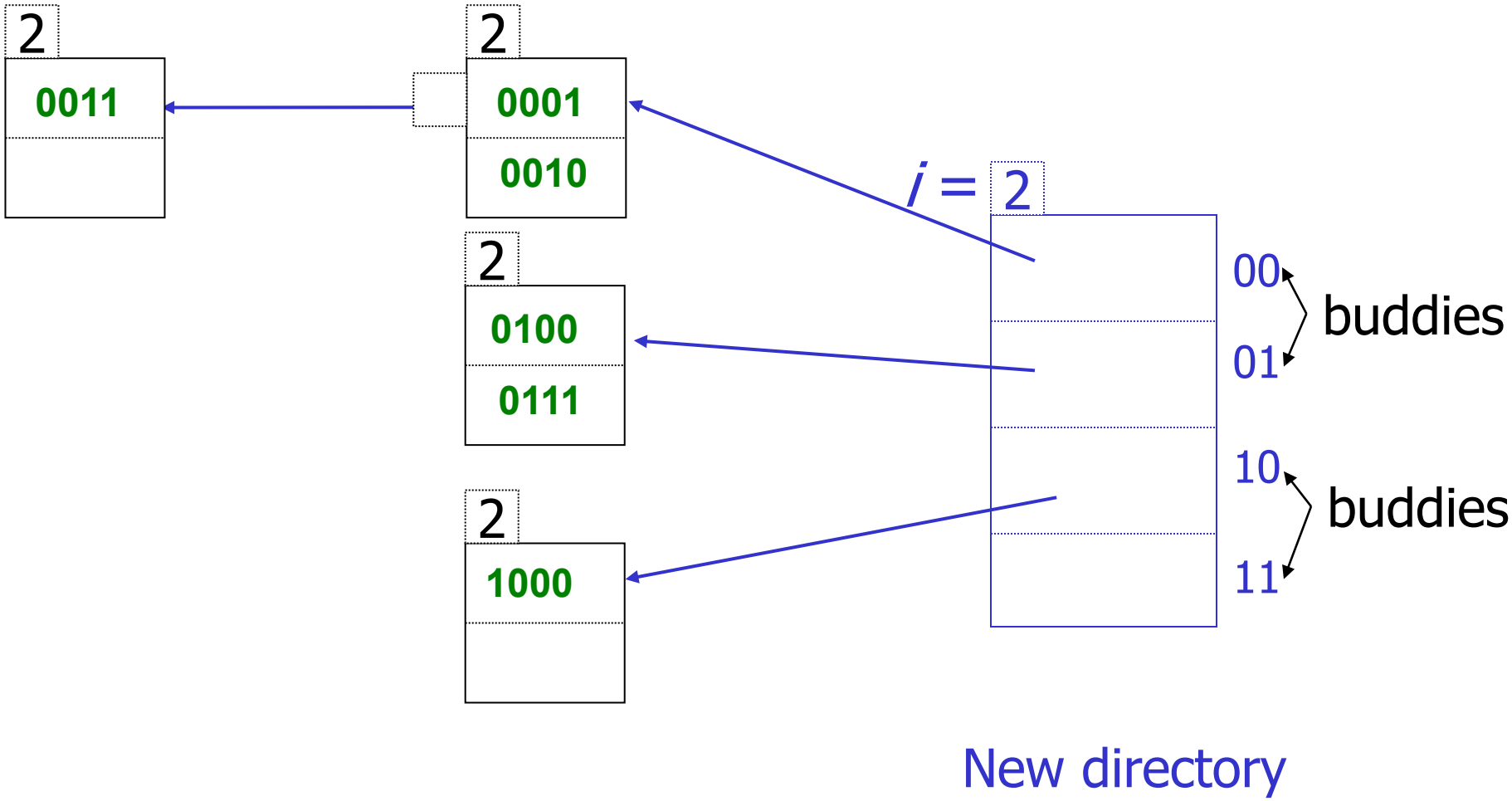
Delete **1001 = 9**

Delete **0110 = 6**



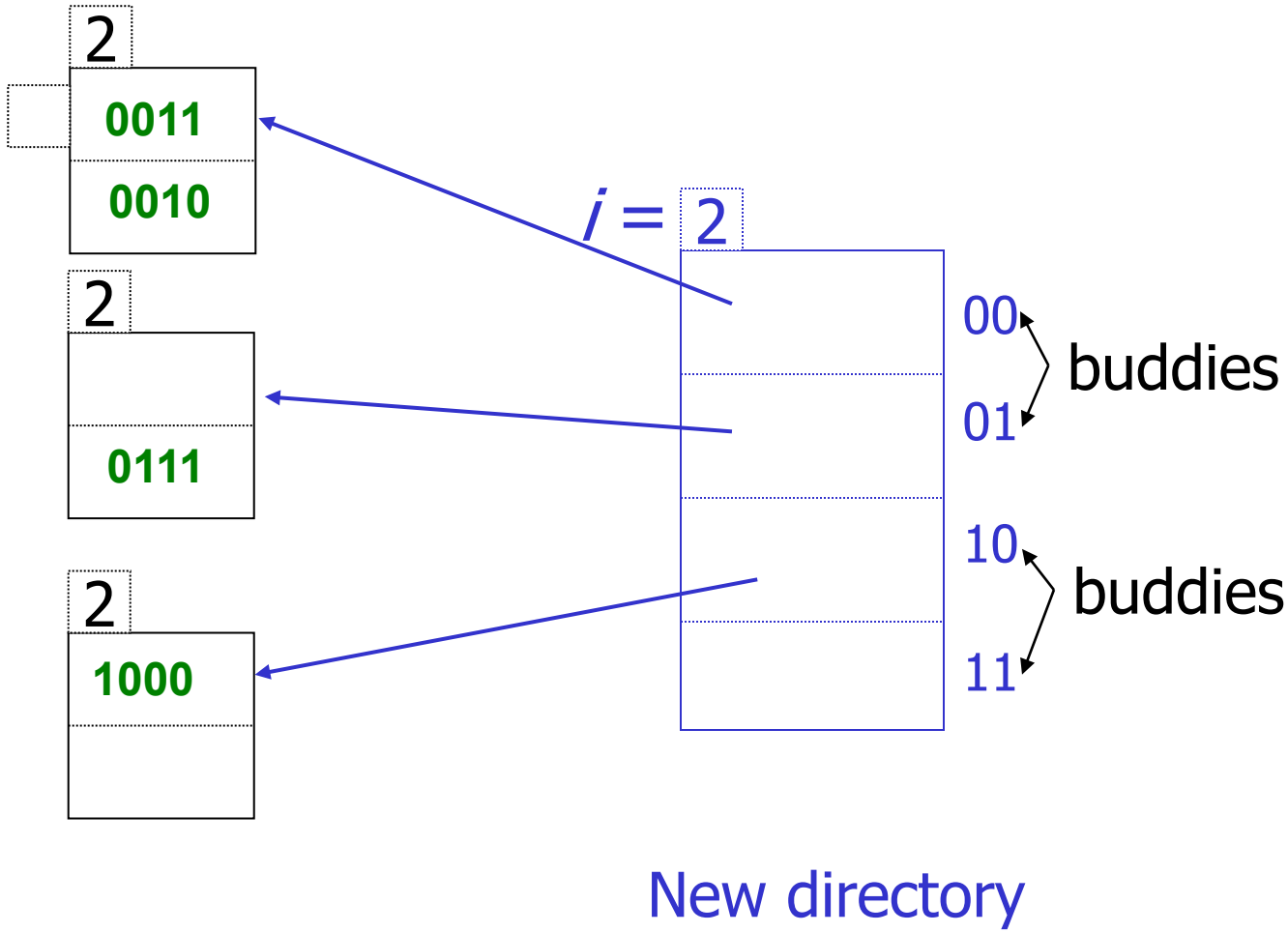
Delete **1001 = 9**

Delete **0110 = 6**



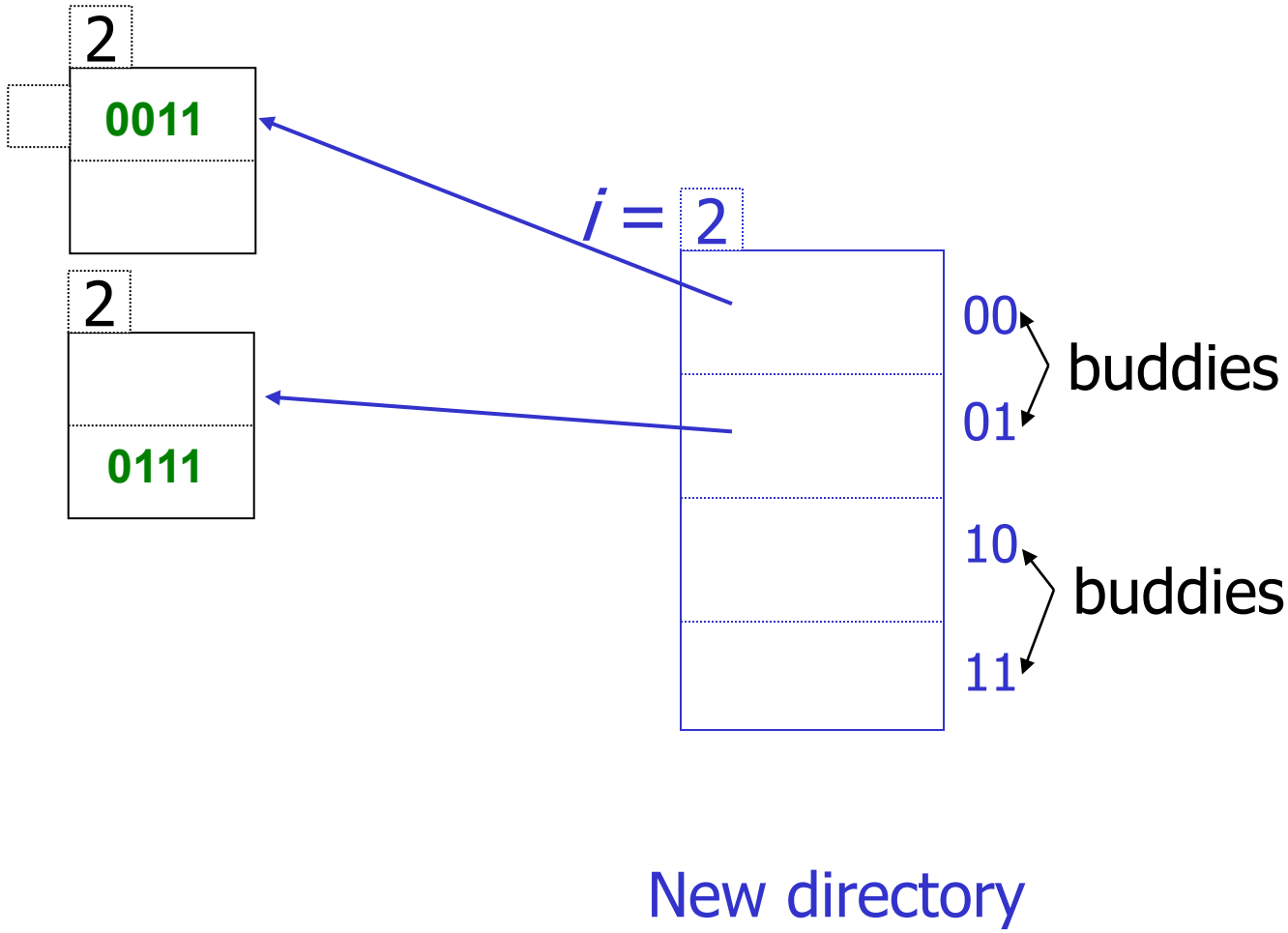
Delete **1001 = 9**

Delete **0110 = 6**



Delete **0100 = 4**

Delete **0001 = 1**



Delete **0010 = 2**

Delete **1000 = 8**

- Problem 10
  - For this problem I have set the $work\_mem = '1.5GB'$
  - Appropriate index: create index worksFor_btree on worksfor USING btree(cname);

Before Any index

	size of relation worksFor integer	Avg execution time to scan(in ms) numeric
1	1000	41.857
2	10000	41.348
3	100000	50.180
4	1000000	138.256

After B+ Tree index

Data Output | Explain | Messages | Notifications | Scratch Pa

	size of relation worksFor integer	Avg execution time to scan(in ms) numeric
1	1000	46.995
2	10000	46.988
3	100000	47.260
4	1000000	100.030

After Hash Index

Data Output | Explain | Messages | Notifications | Scratch Pad

	size of relation worksFor integer	Avg execution time to scan(in ms) numeric
1	1000	46.582
2	10000	47.052
3	100000	61.007
4	1000000	104.000

- Problem 11
  - Appropriate index: create index worksFor_salary_hash on worksfor using hash(salary);

Before Any index

Data Output | Explain | Messages | Notifications | Scratch Pad

	size of relation worksFor integer	Avg execution time to scan(in ms) numeric
1	1000	34.374
2	10000	57.882
3	100000	36.252
4	1000000	78.448

After B+ Tree index

Data Output | Explain | Messages | Notifications | Scratch Pa

	size of relation worksFor integer	Avg execution time to scan(in ms) numeric
1	1000	126.881
2	10000	142.905
3	100000	170.944
4	1000000	361.849

After Hash Index

	size of relation worksFor integer	Avg execution time to scan(in ms) numeric
1	1000	133.368
2	10000	142.187
3	100000	143.561
4	1000000	222.658

- Problem 12
  - Appropriate index:
    - create index worksFor_hash on worksfor using hash(cname);
    - create index worksFor_salary_hash on worksfor using hash(salary);

Before Any index

Data Output | Explain | Messages | Notifications | Scratch Pad

	size of relation worksFor integer	Avg execution time to scan(in ms) numeric
1	1000	86.075
2	10000	59.432
3	100000	61.236
4	1000000	126.713

After B+ Tree index

Data Output | Explain | Messages | Notifications | Scratch Pad

	size of relation worksFor integer	Avg execution time to scan(in ms) numeric
1	1000	0.027
2	10000	0.041
3	100000	0.049
4	1000000	0.081

After Hash Index

Data Output | Explain | Messages | Notifications | Scratch Pad

	size of relation worksFor integer	Avg execution time to scan(in ms) numeric
1	1000	0.028
2	10000	0.011
3	100000	0.034
4	1000000	0.038

- Problem 13
  - Appropriate index:
    - create index worksFor_hash on worksfor using hash(cname);
    - create index worksFor_pid_hash on worksfor using hash(pid);

Before Any index	After B+ Tree index	After Hash Index

Data Output | Explain | Messages | Notifications | Scratch Pad

	size of relation worksFor integer	Avg execution time to scan(in ms) numeric
1	1000	48.583
2	10000	65.926
3	100000	122.843
4	1000000	630.405

Data Output | Explain | Messages | Notifications | Scratch Pad

	size of relation worksFor integer	Avg execution time to scan(in ms) numeric
1	1000	90.521
2	10000	94.966
3	100000	113.782
4	1000000	364.257

Data Output | Explain | Messages | Notifications | Scratch Pad

	size of relation worksFor integer	Avg execution time to scan(in ms) numeric
1	1000	46.751
2	10000	58.878
3	100000	61.795
4	1000000	98.926