1. (Ex 11.1) The last entry that was inserted into the extendible array index can be any of the entries. For any entry, we can find a sequence of inserts with that entry at least that produces the configuration. The last entry inserted doesn’t cause a split because:

- If it belongs to buckets B, C, D, those buckets still have enough free slots. On the other hand, those buckets don’t have split image.
- If it belongs to bucket A or A₂, then in order to be the result of a split, the total entries in A and A₂ must be 5, not 6.

2. Insert 68 causes a split

```
0000 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
```

- A: 3 64 16
- B: 2 5 21
- C: 2 10
- D: 2 15 51
- A₂: 4 20 36 68
- A₂₂: 4 12
2. (Ex 12.2): Just consider I/O cost.

1. \( 6 \leq \rho \leq 50,000 \) (K)

**Sorted File:**

- Locate the (first) page containing the first record satisfying the condition takes a constant time of computation.
- Load that page takes 1 I/O.
- Then we load subsequent pages take 1 I/O for each page.
- We load \( \rho \) in total 50,000 pages \( \Rightarrow \) \( 50,000 \) I/Os.

**Clustered B+ Tree:**

- Locate the first data page takes \( \log_{500,000} \rho \) I/Os
- Where \( F \) is fan-out of \( B+ \) tree.
- \( 500,000 \) is the total number of pages.
- When \( F = 100 \) \( \Rightarrow \) locate first page take 3 I/O.
- Loading subsequent pages take 1 I/O each page \( \Rightarrow \) we load \( 4999 \) subsequent pages.
- In total: \( 5002 \) I/Os.

**Linear Hashed Index:**

- Hash index is no help with condition that is not equality.
- Scan the hash index is prohibitively expensive.

\( \Rightarrow \) Sorted File is cheapest.
Sorted File:
If we use binary search we can compute the page containing the record (just 1 since R.a is a candidate key) in $\log_2 500,000 = 19$ I/Os.
(Assume pages have continuous page IDs).
However, if we take into account the fact that each page has 10 records and values of R.a ranges from 0 to 4,999,999 then we can locate the page to be page 5000-th. And the task is done in just 1 I/O.

B+ Tree:
Takes $\log_2 500,000 = 3$ I/Os (typical case)

Hashed Index:
It takes about 1.2 I/O to retrieve the (linear) hash bucket
It takes 1 more I/O to retrieve the data page
$\Rightarrow$ in total 2.2 I/Os.

$\Rightarrow$ The cheapest plan is Sorted file or linear hash index (depend on the search algorithm used in Sorted file).
Sorted File:
Locate the page that contains the first record with \( a > 50,000 \) and \( a < 50,010 \) (R).

\[ \log_2 50,000 = 19 \] I/Os if we binary search
or 1 I/O if we compute directly; i.e., it must be the page \( \geq 50,000 \).

All records that satisfy \( a > 50,000 \) and \( a < 50,010 \) are in the same page, thus no additional I/O is required. Create the last record of that page has \( a = 50,010 \) and it confirms that no more record can be found.

B+ tree:
Locate the first page takes \( \log_2 500,000 = 3 \) I/Os.

No additional page is needed.

Hashed index:
Hash index doesn't help with this query.
Scan hash index is prohibitively expensive.

Cheapest plan is Sorted File or B+ tree, depending on the search algorithm used in Sorted File.
4. \( a \geq 50,000 \) (R)

Sorted File:
Scan the whole file: takes 500,000 I/Os.

Bt tree:
Get to the leftmost data pages takes
\[ \log_{10} 500,000 = 19 \text{ I/Os} \]
Scan the whole file via the linked list takes
500,000 I/Os
\[ \Rightarrow \text{in total } 500,019 \text{ I/Os}. \]

Hashed index:
Scan the index is prohibitively expensive.
The total number of I/Os depends on
the hash function and other factors. In worst case, each data entry index data entry can cause an I/O. Caching pages can reduce the
number of I/Os but it is still very large.

\[ \Rightarrow \text{cheapest plan: Sorted File.} \]

3. (Ex 15.6.2)

\[ a \subseteq (\pi_c (R \times S)) \]

Clearly, attributes involved in \( c \) must be a subset of attributes involved in \( e \).

Assume \( R \) has 3 attributes \( r_1, r_2, r_3 \)
\( S \) has 3 attributes \( s_1, s_2, s_3 \).
a. \( \sigma_c (\pi_x (R \times S)) = \pi_{l_1} (\sigma_{c_1} (R) \times S) \)

Then:

\[ l = c \]

\[ c = c_1 \]

\( c_1 \) involves only attributes of \( R \)

\( l_1 \) has all attributes of \( R \) that appear in \( c_1 \)

\((*)\) otherwise we can push projection of \( R \) on those attributes.

Example:

\( c_1 \) involves \( r_2 \)

\( l_1 \) involves \( r_2 r_3 r_5 \)

\[ \sigma_{r_4} (\pi_{r_4 r_2 r_3} (R \times S)) = \pi_{l_1} (\sigma_{c_1} (R) \times \sigma_{c_2} (S)) \]

b. \( \sigma_c (\pi_x (R \times S)) = \pi_{l_1} (\sigma_{c_1} (R) \times \sigma_{c_2} (S)) \)

Then:

\[ c = c_1 \land c_2 \]

\( c_1 \) involves attributes of \( R \) only

\( c_2 \) involves attributes of \( S \) only

\[ l = l_1 \]

If the attributes of \( R \) appear in \( l_1 \) is just a subset of all \( R \)'s attributes, then we can push the projection ahead, i.e. perform projection before the Cartesian product. Thus \( l_1 \) must have all attributes of \( R \). (\( l_1 \) can't have no attributes of \( R \) since \( c_1 \subset c_2 \)).
The same for $S$, i.e. $l_1$ has all attributes of $S$.

**Example:**
- $C_1$ involves $r_2$
- $C_2$ involves $r_3$
- $C_3$ involves all attributes of $R$ and $S$

$\bar{C}_1 = r_2 \land r_3 \land r_4 \land r_5 \land (R \times S))$

$\equiv \prod_{r_1 \neq r_2} \prod_{r_3 \neq r_4} \prod_{r_5 \neq r_3} (\bar{C}_1 (R) \times \bar{C}_2 (S))$

**c.** $\bar{C}_C (\Pi_C (R \times S)) \equiv \bar{C}_C (\Pi_{r_1} (\Pi_{r_2} (R) \times S))$

Then:
- $l_2 = l$
- $l_2$ involves only $R$
- $C_1$: all conjuncts in $C$ involve both $R$ and $S$; otherwise we can push the conjunct ahead.

Since we don’t push projection of $S$ ahead, $l_4$, either:
- $l_4$ contains all attributes of $S$
- $l_4$ contains no attribute of $S$. This doesn’t make sense because if that is the case, $l_4$ contains only attributes of $R$ and $l_4$ must be subset of $l_2$, and thus we just need $l_3$, no need for $l_4$, i.e. the equivalent expression is $\bar{C}_C (\Pi_{r_1} (R) \times S)$.

For the attributes of $R$ appear in $l_3$, they must also appear in $l_2$, i.e. $(l_2 \cap R) \subseteq l_2$. Also, $l \cap R = l_2$, otherwise we should $\Pi_{l \cap R} (R)$ instead of $\Pi_{l_2} (R)$.

**Example:**
- $l_2$ involves $r_2$
- $l_4$ involves $r_1$, $r_2$, $r_3$, $r_4$
- $C_1$ involves $r_2$, $r_3$

$\bar{C}_2 = r_1 (\Pi_{r_2} (R \times S)) \equiv \bar{C}_2 = r_1 (\Pi_{r_1} (R \times S))$. 


Security / Access Control

1. At time 100, Bob issues execute
   REVOKE SELECT ON T FROM CHRIS

   Then the graph will be:

   ![Graph 1](image1.png)

   1. True
   2. True

2. At time 100 Ann: REVOKE SELECT ON T FROM Jim
   At time 110 Chris: REVOKE SELECT ON T FROM Jim

   The graph after that will be:

   ![Graph 2](image2.png)

   1. True
   2. False
   3. Sue, Jim, Ann, Bob. (in old graph)

   In new graph, Dave doesn't have select privilege anymore.