CS302 - Data Structures *using C++*

Topic: Processing Data in External Storage

Kostas Alexis

Look at External Storage

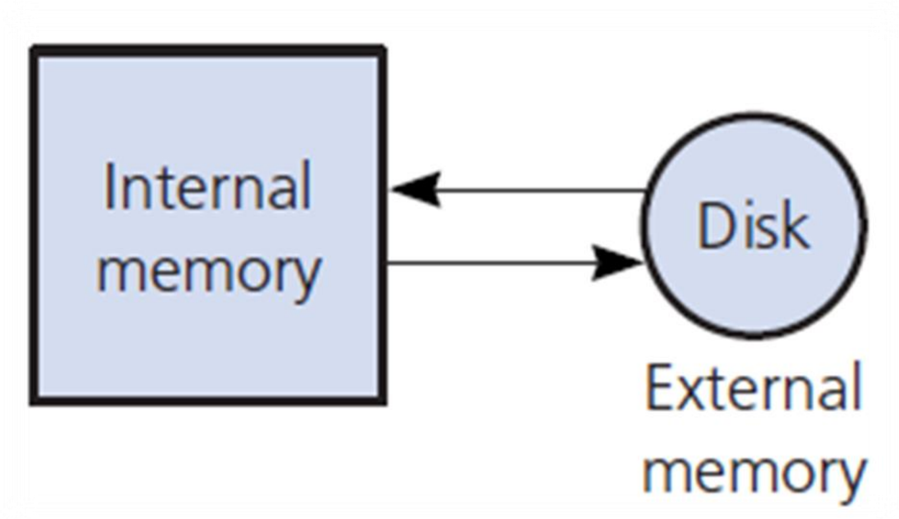
- External storage
  - Used when program reads/writes data to/from a C++ file
- Generally there is more external storage than internally memory
- Direct access files essential for external data collections

Look at External Storage

- External storage
  - Used when program reads/writes data to/from a C++ file
- **Generally there is more external storage than internally memory**
- Direct access files essential for external data collections

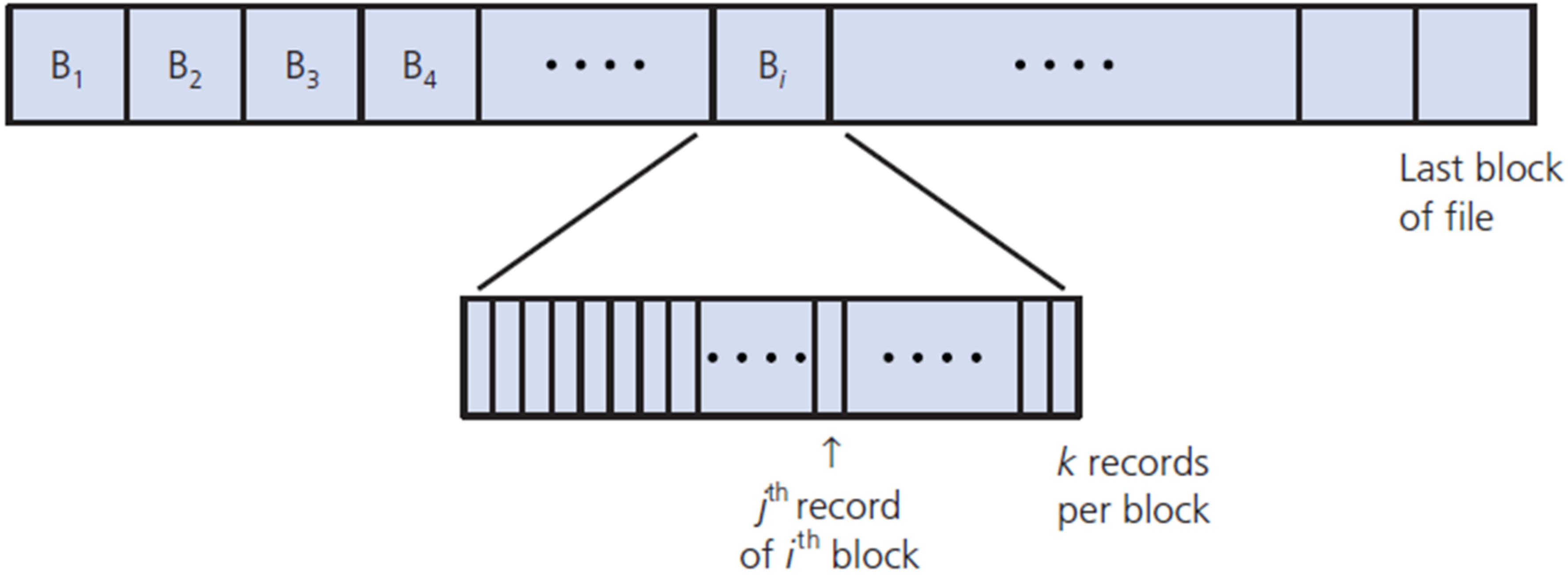
Look at External Storage

- Internal and external memory



Look at External Storage

- A file partitioned into blocks of records



Look at External Storage

- Direct access I/O
  - Involves blocks instead of records
- Buffer stores data (blocks) temporarily
- Record updated within block (in buffer)
- Work to minimize block I/O
  - Takes more time for disk access

Working with External Data

- Sorting data in an external file
    - In principle we look to use a sorting algorithm

Working with External Data

- Sorting data in an external file
  - In principle we look to use a sorting algorithm
  - **Problem is that not all data are in the internal memory – e.g., not all of them are in an array**

Working with External Data

- Sorting data in an external file
  - In principle we look to use a sorting algorithm
  - **Problem is that not all data are in the internal memory – e.g., not all of them are in an array**
  - We will use a modified version of mergeSort to overcome this problem

Working with External Data

- Sorting data in an external file
  - In principle we look to use a sorting algorithm
  - **Problem is that not all data are in the internal memory – e.g., not all of them are in an array**
  - We will use a modified version of mergeSort to overcome this problem
- The basis of mergeSort is that you can easily merge two sorted segments of data records into a third sorted segment that is the combination of the two.

Working with External Data

- Sorting data in an external file
  - In principle we look to use a sorting algorithm
  - **Problem is that not all data are in the internal memory – e.g., not all of them are in an array**
  - We will use a modified version of mergeSort to overcome this problem
- The basis of mergeSort is that you can easily merge two sorted segments of data records into a third sorted segment that is the combination of the two.
  - If S_1 and S_2 are sorted segments of records, the first step of the merge is to compare the first record of each segment and select the record with the smallest sort key.
  - If the record from S_1 is selected, the next step is to compare the second record of S_1 to the first record of S_2.
  - This process is continued until all of the records have been considered.

Working with External Data

- Sorting data in an external file
  - In principle we look to use a sorting algorithm
  - **Problem is that not all data are in the internal memory – e.g., not all of them are in an array**
  - We will use a modified version of mergeSort to overcome this problem
- The basis of mergeSort is that you can easily merge two sorted segments of data records into a third sorted segment that is the combination of the two.
  - If S_1 and S_2 are sorted segments of records, the first step of the merge is to compare the first record of each segment and select the record with the smallest sort key.
  - If the record from S_1 is selected, the next step is to compare the second record of S_1 to the first record of S_2.
  - This process is continued until all of the records have been considered.
  - **The key observation is that at any step, the merge never needs to look beyond the *leading edge* of either segment.**

Working with External Data

- *Example problem:* An external file contains 1,600 employee records. You want to sort these records by Social Security Number. Each block contains 100 records, and thus the file contains 16 blocks B_i. Assume that the program can access only enough internal memory to manipulate about 300 records (three blocks) at a time.

Working with External Data

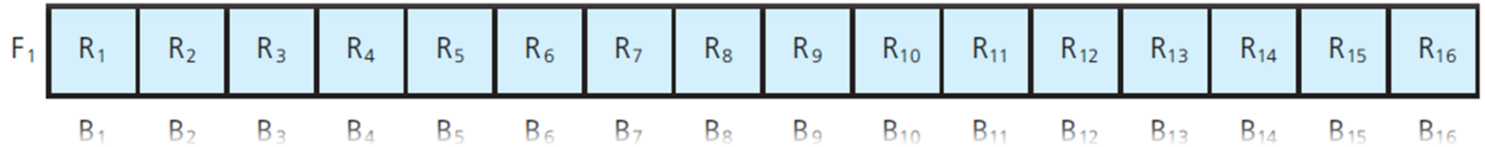
- *Example problem:*
  - An external file contains 1,600 employee records.
  - You want to sort these records by Social Security Number.
  - Each block contains 100 records, and thus the file contains 16 blocks B_i.
  - Assume that the program can access only enough internal memory to manipulate about 300 records (three blocks) at a time.

Working with External Data

- Sorting operation will be organized around two phases:

Working with External Data

- Sorting operation will be organized around two phases:
  - **Phase 1:** Read a block from file F into internal memory, sort its records by using an internal sort, and write the sorted block out to work file F_1 before you read the next block from F. After you process all (16) blocks of F, F_1 contains (16) sorted runs R_i. That is that F_1 contains (16) blocks of records with the records within each block sorted among themselves.

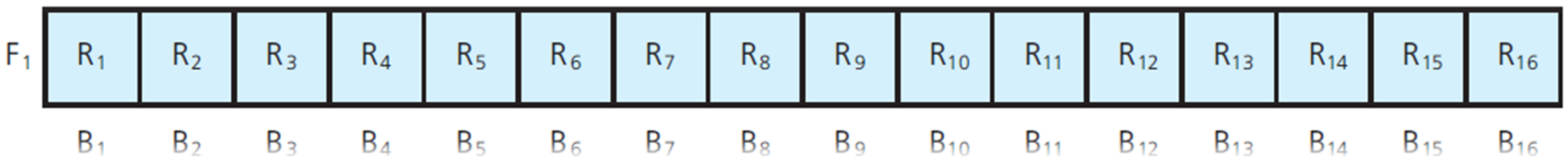


(a) 16 sorted runs, 1 block each, in file *F_1*

Working with External Data

- Sorting operation will be organized around two phases:
    - **Phase 1:**
        - Read a block from file **F** into internal memory, sort its records by using an internal sort, and write the sorted block out to work file **F_1** before you read the next block from **F**.
        - After you process all (16) blocks of **F, F_1** contains (16) sorted runs **R_i**. That is that **F_1** contains (16) blocks of records with the records within each block sorted among themselves.

(a) 16 sorted runs, 1 block each, in file F_1


F1
R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15 R16
B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 B16

Working with External Data

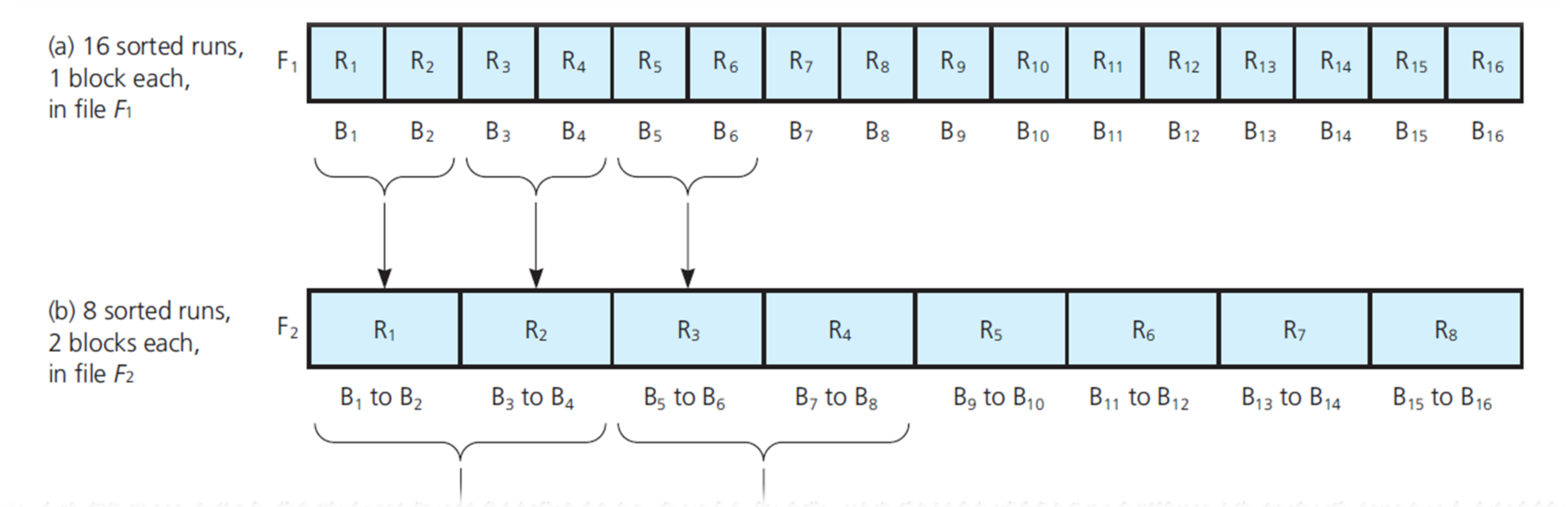
- Sorting operation will be organized around two phases:
  - **Phase 1:** Read a block from file F into internal memory, sort its records by using an internal sort, and write the sorted block out to work file F_1 before you read the next block from F. After you process all (16) blocks of F, F_1 contains (16) sorted runs R_i. That is that F_1 contains (16) blocks of records with the records within each block sorted among themselves.
  - **Phase 2:**
    - Phase 2 is a sequence of merge steps.
    - Each merge step merges pairs of sorted runs to form larger sorted runs.
    - With each merge step, the number of blocks in each sorted run doubles, and thus the total number of sorted runs is halved.
    - The final step merges the two sorted runs into one, which is written to $\mathbf{F_1}$. At this point, $\mathbf{F_1}$ will contain all of the records of the original file in sorted order.
    - Work file $\mathbf{F_2}$ has executed its intermediate role.

Working with External Data

- Sorting operation will be organized around two phases:
  - **Phase 1:** Read a block from file F into internal memory, sort its records by using an internal sort, and write the sorted block out to work file F_1 before you read the next block from F. After you process all (16) blocks of F, F_1 contains (16) sorted runs R_i. That is that F_1 contains (16) blocks of records with the records within each block sorted among themselves.
  - **Phase 2:** Phase 2 is a sequence of merge steps. Each merge step merges pairs of sorted runs to form larger sorted runs. With each merge step, the number of blocks in each sorted run doubles, and thus the total number of sorted runs ins halved. The final step merges the two sorted runs into one, which is written to F_1. At this point, F_1 will contain all of the records of the original file in sorted order. Work file F_2 has executed its intermediate role.

Working with External Data

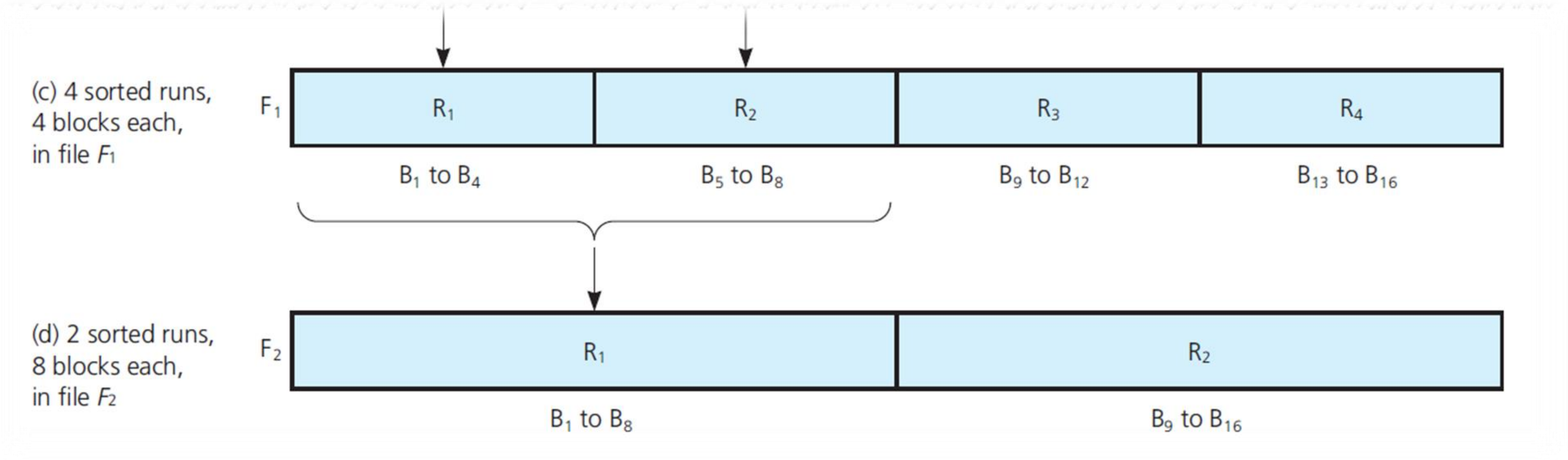
- Sorting a block of an external file F by merging the results sorts and using two external work files F_1 and F_2.



(a) 16 sorted runs, 1 block each, in file F_1
F_1
R_1 R_2 R_3 R_4 R_5 R_6 R_7 R_8 R_9 R_{10} R_{11} R_{12} R_{13} R_{14} R_{15} R_{16}
B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 B_{10} B_{11} B_{12} B_{13} B_{14} B_{15} B_{16}
(b) 8 sorted runs, 2 blocks each, in file F_2
F_2
R_1 R_2 R_3 R_4 R_5 R_6 R_7 R_8
B_1 to B_2
B_3 to B_4
B_5 to B_6
B_7 to B_8
B_9 to B_{10}
B_{11} to B_{12}
B_{13} to B_{14}
B_{15} to B_{16}

Working with External Data

- Sorting a block of an external file F by merging the results of internal sorts and using two external work files F_1 and F_2.



(c) 4 sorted runs, 4 blocks each, in file *F*1
F1
R1
R2
R3
R4
B1 to B4
B5 to B8
B9 to B12
B13 to B16
(d) 2 sorted runs, 8 blocks each, in file *F*2
F2
R1
R2
B1 to B8
B9 to B16

Working with External Data

- How to merge the sorted runs at each step of phase 2?
  - Problem statement provides only sufficient internal memory to manipulate at most (300) records at once.
  - However, in the later steps of phase 2, runs contain more than 300 records each, so you must merge the runs a piece at a time.

Working with External Data

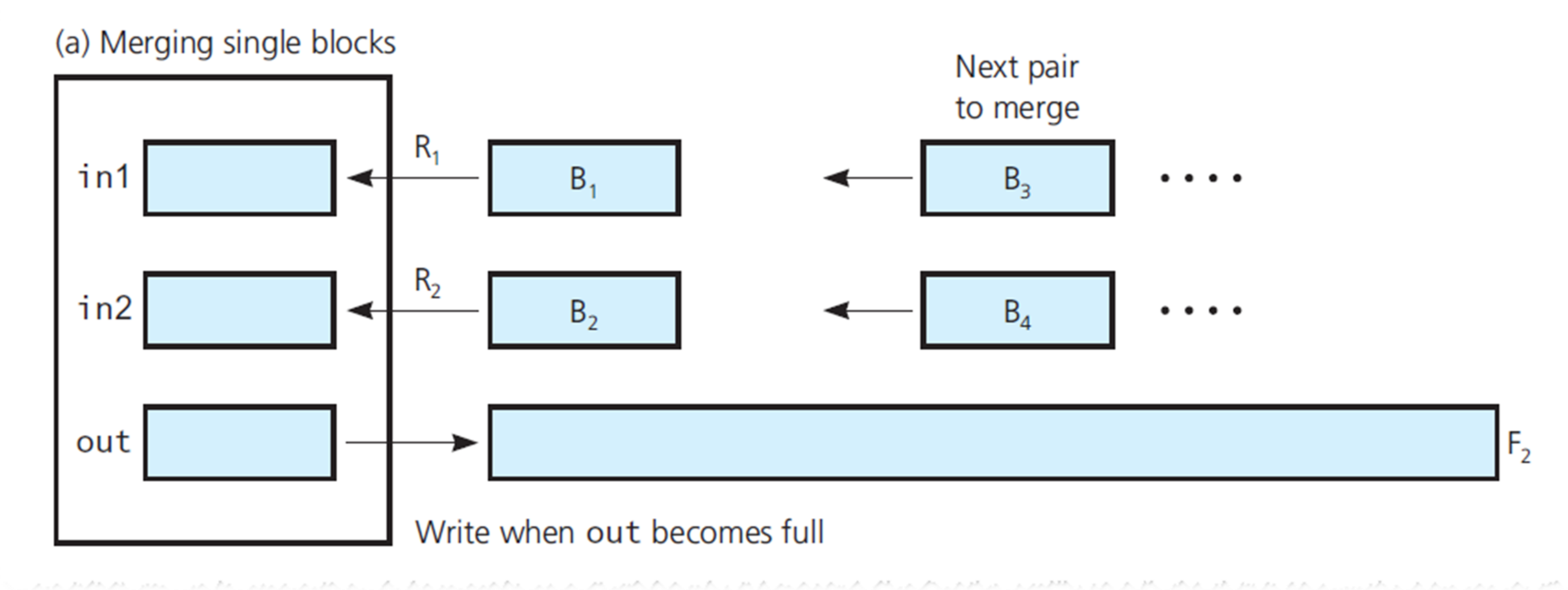
- How to merge the sorted runs at each step of phase 2?
  - Problem statement provides only sufficient internal memory to manipulate at most (300) records at once.
  - However, in the later steps of phase 2, runs contain more than 300 records each, so you must merge the runs a piece at a time.
  - **To accomplish this merge, you must divide the program's internal memory into three arrays, `in1`, `in2` and `out` each capable of holding (100)records (the block size).**

Working with External Data

- How to merge the sorted runs at each step of phase 2?
    - Problem statement provides only sufficient internal memory to manipulate at most (300) records at once.
    - However, in the later steps of phase 2, runs contain more than 300 records each, so you must merge the runs a piece at a time.
    - **To accomplish this merge, you must divide the program’s internal memory into three arrays, `in1`, `in2` and `out` each capable of holding (100)records (the block size).**
        - You read block-sized pieces of the runs into `in1` and `in2` and merge them into the array out.
        - Whenever an `in` array is exhausted – that is, when all of its entries have been copied to `out` – you read the next piece of the run into an `in` array.
        - Whenever the `out` array becomes full, you write the complete piece of the new sorted runs to one of the files.

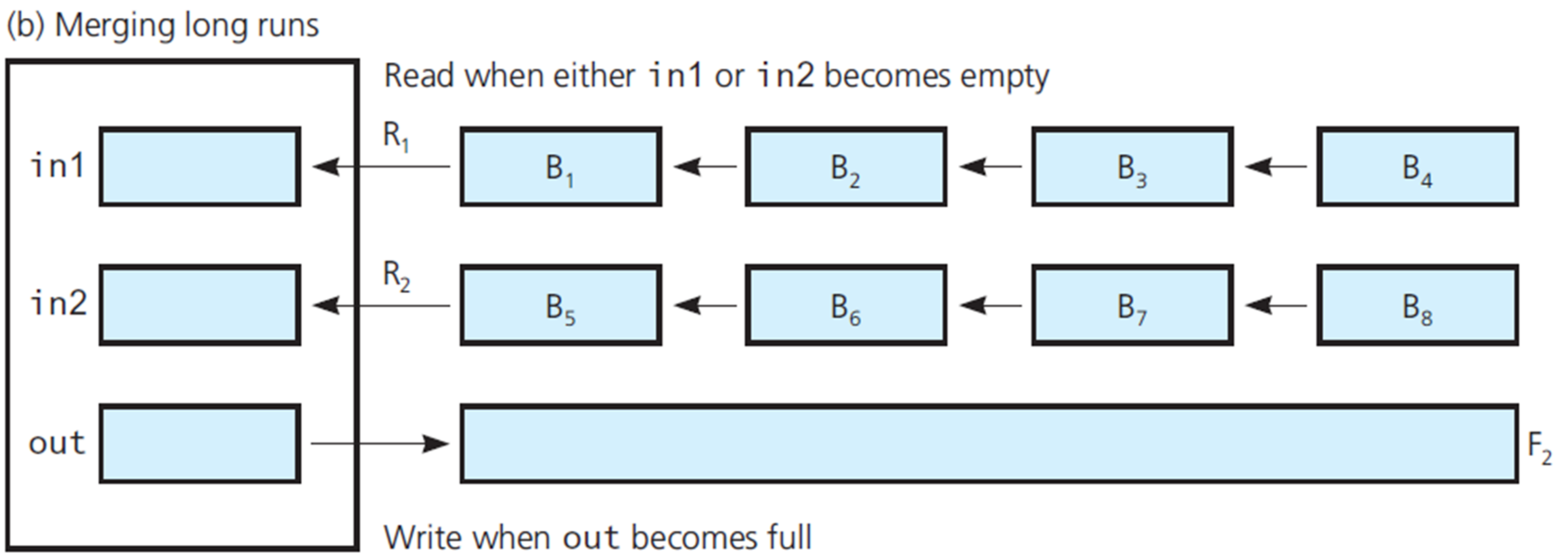
Working with External Data

- Phase 2 of an external sort: Merging sorted runs



Working with External Data

- Phase 2 of an external sort: Merging sorted runs


(b) Merging long runs
Read when either in1 or in2 becomes empty
in1
in2
out
R₁
R₂
B₁
B₂
B₃
B₄
B₅
B₆
B₇
B₈
F₂
Write when out becomes full

Working with External Data

- externalMergesort

```
externalMergesort(unsortedFilename: string, sortedFilename: string)
{
        Associate unsortedFileName with the file variable inFile and sortedFilename with outFile
        // Phase 1: Sort file block by block and count the blocks
        blockSort(inFile,tempFile1,numberOfBlocks)
        // Phase 2: Merge runs of size 1,2,4,8,…,numberOfBlocks/2
        // (uses two temporary files and a toggle that tracks files for each merge step)
        toggle = 1
        for (size = 1 through numberOfBlocks/2 with increments of size)
        {
                if (toggle == 1)
                        mergeFile(tempFile1, tempFile2, size, numberOfBlocks)
                else
                {
                        mergeFile(tempFile2, tempFile1, size, numberOfBlocks)
                        toggle = -toggle
                }
        }
        // Copy the current temporary file to outFile
        if (toggle==1)
                copyFile(tempFile1, outFile)
        else
                copyFile(tempFile2, outFile)
}
```

Working with External Data

- externalMergesort

```
blockSort(inFile: File, outFile: File, numberOfBlocks: integer)
{
        prepare inFile for input
        prepare outFile for output
        numberOfBlocks = 0
        while (more blocks in inFile remain to be read)
        {
                numberOfBlocks++
                buffer.readBlock(inFile,numberOfBlocks)
                sortArray(buffer) // use some internal sort method
                buffer.writeBlock(outFile, numberOfBlocks)
        }
        close inFile and outFile
}
```

Working with External Data

- externalMergesort

```
// Merge blocks from one file to another
// Precondition: nFile is an external file that contains numberOfBlocks sorted blocks organized into
// runs of runSize blocks each.
// Postcondition: outFile contains the merged runs of inFile. Both files are closed.
// Calls: mergeRuns
mergeFile(inFile: File, outFile: File, runSize: integer, numberOfBlocks)
{
        prepare inFile for input
        prepare outFile for output
        for (next = 1 to numberOfBlocks with increments of 2*runSize)
        {
                // Invariant: Runs in outFile are ordered
                mergeRuns(inFile, outFile, next, runSize)
        }
        close inFile and outFile
}
```

Working with External Data

- externalMergesort

```
// Merges two consecutive sorted runs in a file.
// Precondition: fromFile is an external file of sorted runs
// open for input. toFile is an external file of sorted runs.
// open for output. start is the block nuber of the first run on fromFile to be merged. This run
// contains size blocks.
// Run 1: Block start to block start + size – 1
// Run 2: Block start + size to start + (2*size) – 1
// Postcondition: The merged runs from fromFile are appended to toFile. The files remain open.
mergeRuns(fromFile: File, toFile: File, start: integer, size: integer)
{
        // Initialize the input buffers for runs 1 and 2
        in1.readBlock(fromFile, first block of Run 1)
        in2.readBlock(fromFile, first block of Run 2)
        // Merge until one of the runs is finished Whenever an input buffer is exhauster, the next
        // block is read. Whenever the output buffer is full, it is written.
        while (neither run is finished)
        {
                // Invariant: out and each block in toFile are ordered
                Select the smallest “leading eldge” of in1 and in2 and place in the next position of out
                if (out is full)
                        out.writeBlock(toFile, next block of toFile)
                if (in1 is exhausted and blocks remain in Run 1)
                        in1.readBlock(fromFile, next block of Run 1)
                if (in2 is exhausted and blocks remain in Run 2)
                        in2.readBlock(fromFile, next block of Run 2)
        }
```

Working with External Data

- externalMergesort

```
        // Assertion: Exactly one of the runs is complete

        // Append the remainder of the unfinished input buffer to the output buffer and write it
        while (in1 is not exhausted)
                // Invariant: out is ordered
                Place next item of in1 into the next position of out
        while (in2 is not exhausted)
                // Invariant: out is ordered
                Place next item of in2 into the next position of out
        out.writeBlock(toFile, next block of toFile)

        // Finish off the remaining complete blocks
        while (blocks remain in Run 1)
        {
                // Invariant: Each block in toFile is ordered
                in1.readBlock(fromFile, next block of Run 1)
                in1.writeBlock(toFile, next block of toFile)
        }
        while (blocks remain in Run 2)
        {
                // Invariant: Each block in toFile is ordered
                in2.readBlock(fromFile, next block of Run2)
                in2.writeBlock(toFile, next block of toFile)
        }
}
```

Basic Data Management Operations

- Suppose that you have direct access file of records that have search keys like the records we considered in the ADT Dictionary. The file is partitioned into blocks. Imagine that we store the records in order by their search key, perhaps sorting the file by using the external mergeSort. Once sorted, we can traverse the file in sorted order.

Basic Data Management Operations

- Traversal of sorted file (visit method call for each item)

```
Traverse(dataFile: File, numberOfBlocks: integer, recordsPerBlock: integer, visit: FunctionType)
{
        // read each block of file dataFile into an internal buffer buf
        for (blockNumber = 1 through numberOfBlocks)
        {
                buf.readBlock(dataFile, blockNumber)
                // Visit each record in the block
                for (recordNumber = 1 through recordsPerBlock)
                        Visit record buf.getRecord(recordNumber – 1)
        }
}
```

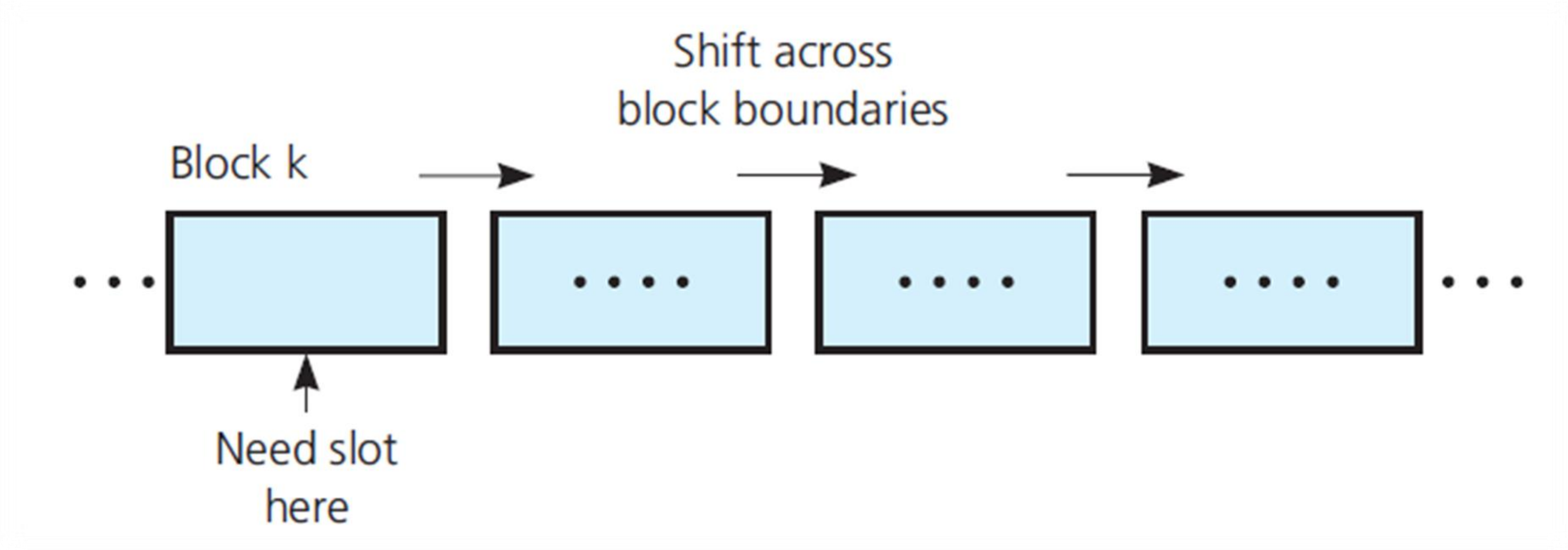
Basic Data Management Operations

- Retrieval of record from the sorted file using a binary search algorithm

```
// Searches blocks first through last of the file dataFile for the record whose key equals searchKey
getItem(dataFile: File, recordsPerBlock: integer, first: integer, last: integer, searchKey: KeyType): ItemType
{
        if (first > last or nothing is left to read from dataFile)
                throw NotFoundException
        else
        {
                // Read the middle block of file dataFile into array buf
                mid = (first + last)/2
                buf.readBlock(dataFile, mid)
                if ((searchKey >= (buf.getRecord(0)).getKey()) &&
                        (searchKey <= (buf.getRecord(recordsPerBlock-1)).getKey()) )
                {
                        // Desired block is found
                        Search buffer buf for record buf.getRecord(j) whose search key equals searchKey
                        if (record is found)
                                return buf.getRecord(j)
                        else
                                throw NotFoundException
                }
                // Else search appropriate half of the file
                else if (searchKey < (buf.getRecord(0)).getKey())
                        return getItem(dataFile, recordsPerBlock, first, mid-1, searchKey)
                else
                        return getItem(dataFile, recordsPerBlock, mid+1, last, searchKey)
        }
}
```

Basic Data Management Operations

- Shifting across block boundaries



- **Costly**

Indexing an External File

- Two of the best approaches to external data management utilize variations of the internal hashing and search-tree schemes.

Indexing an External File

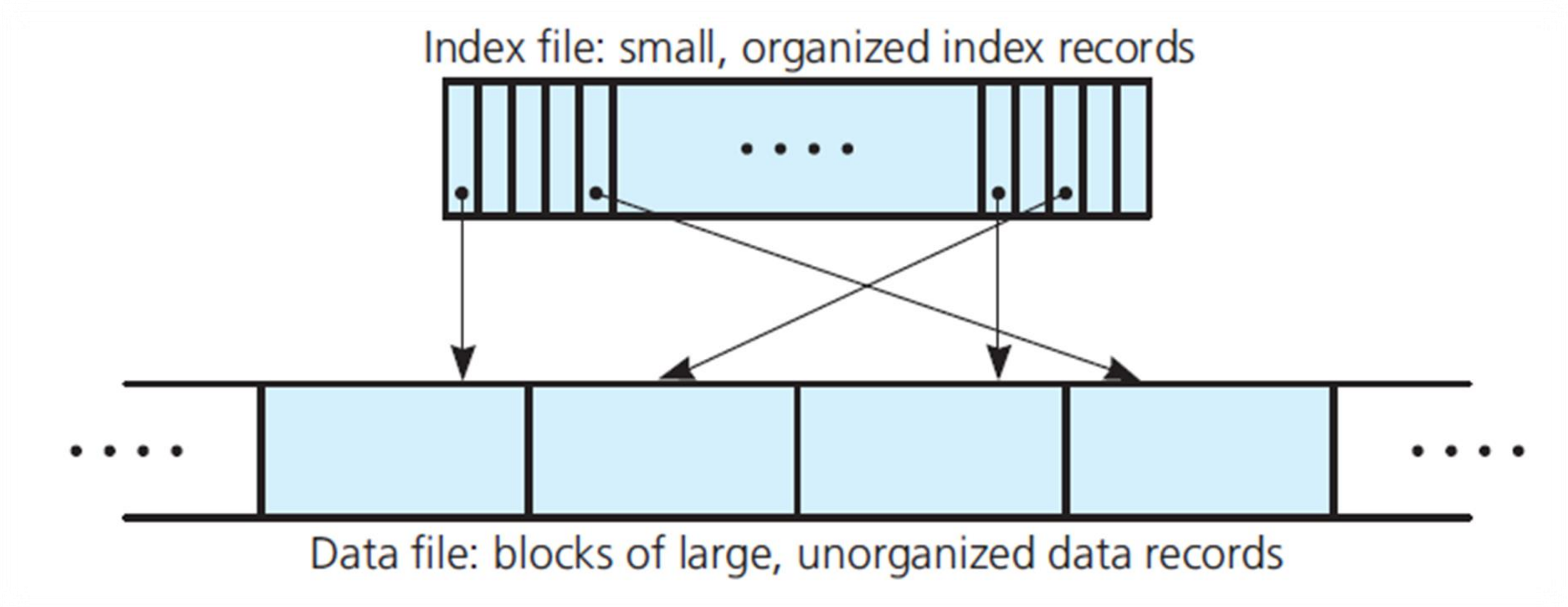
- Two of the best approaches to external data management utilize variations of the internal hashing and search-tree schemes.
    - The biggest difference between the internal and external versions of these approaches is that the external versions often have an index to the data file rather than an organization of the data file itself.
    - An index to a data file is conceptually similar to other indexes.
    - An index to the data file is simply another file, called the **index file**, that contains an index record for each record in the data file.
    - .

Indexing an External File

- Two of the best approaches to external data management utilize variations of the internal hashing and search-tree schemes.
  - The biggest difference between the internal and external versions of these approaches is that the external versions often have an index to the data file rather than an organization of the data file itself.
  - An index to a data file is conceptually similar to other indexes.
  - An index to the data file is simply another file, called the **index file**, that contains an index record for each record in the data file.
  - An **index record** has two parts:
    - a **key**, which contains the same value as the search key of its corresponding record in the data file,
    - a **reference**, which shows the number of the block in the data file that contains this data record.
  - You therefore can locate the block of the data file that contains the record whose search key equals **searchKey** by searching the index file for the index record whose key equals searchKey.

Indexing an External File

- A data file with an index

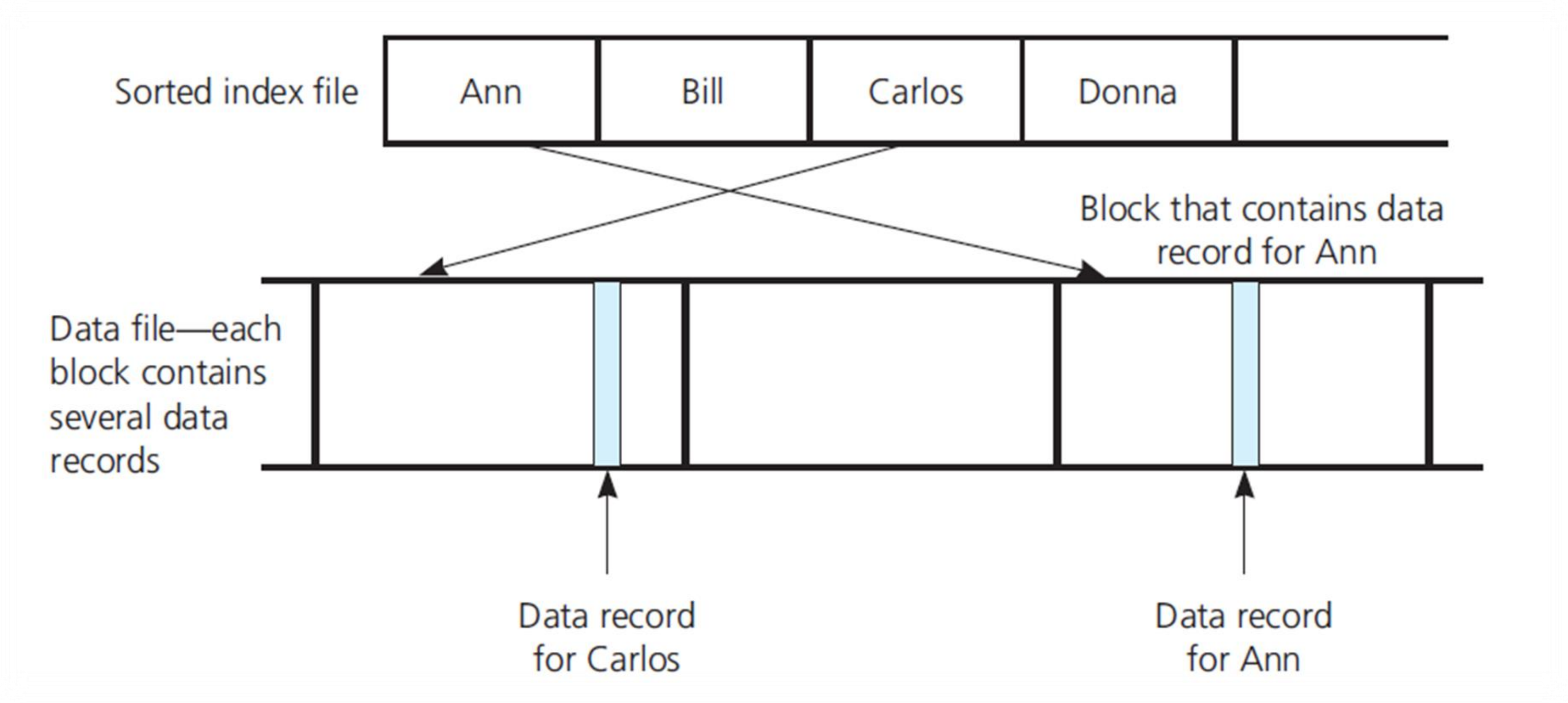

Index file: small, organized index records
Data file: blocks of large, unorganized data records

Indexing an External File

- **Advantages of an index file**
  - Index record smaller than a data record
  - Data file need not be kept in any particular order
  - Possible to maintain several indexes simultaneously
  - Index file reduces number of block accesses
  - Shift index records, not data records

Indexing an External File

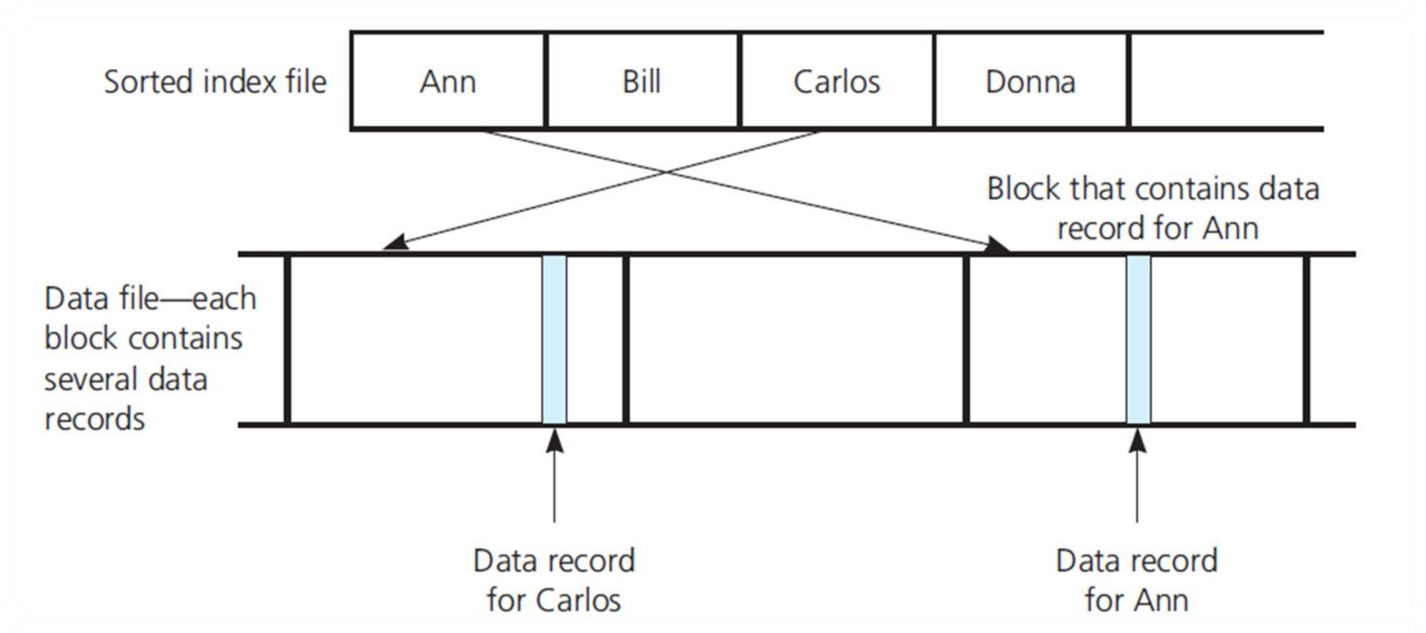
- A data file with a sorted index file



Sorted index file
Ann
Bill
Carlos
Donna
Block that contains data record for Ann
Data file—each block contains several data records
Data record for Carlos
Data record for Ann

Indexing an External File

- Although we do not organize the data file, we must organize the index file so that we can search and update it rapidly.
- Example: how to implement retrieval assuming the index file simply stores the index records sequentially, sorted by their keys as in the Figure.


Sorted index file
Ann
Bill
Carlos
Donna
Block that contains data record for Ann
Data file—each block contains several data records
Data record for Carlos
Data record for Ann

Indexing an External File

- Example: how to implement retrieval assuming the index file simply stores the index records sequentially, sorted by their keys as in the Figure.

```
// Searches the file dataFile for the record whose search key equals searchKey.
// Returns the record if found, else throws NotFound Exception.
getItem(indexFile: File, dataFile: File, searchKey: KeyType): ItemType
{
        if (no blocks are left in indexFile to read)
                throw NotFoundException
        else
        {
                // Read the middle block of indexfile into object buf
                mid = number of middle block of indexFile
                buf.readBlock(indexFile, mid)
                if ((searchKey > (buf.getRecord(0)).getKey()) &&
                        (searchKey <= (buf.getRecord(indexrecordsPerBlock – 1)).getKey()))
                {
                        // Desired block of index file found
                        Search buf for index file record whose key value equals searchKey
```

Indexing an External File

- Example: how to implement retrieval assuming the index file simply stores the index records sequentially, sorted by their keys as in the Figure.

```
            if (index record buf.getRecord(j) is found)
            {
                blockNum = number of the data-file block to which
                    buf.getRecord(j) points
                data.readBlock(dataFile, blockNum)
                Find data record data.getRecord(k) whose search key equals
                    searchKey
                return data.getRecord(k)
            }
            else
                throw NotFoundException
        }
    }
    // Else search appropriate half of index file
    else if (searchKey < (buf.getRecord(0)).getKey())
        return getItem(first half of indexFile, dataFile, searchKey)
    else
        return getItem(second half of indexFile, dataFile, searchKey)
    }
}
```

Indexing an External File

- Because the index records are far smaller than the data records, the index file contains far fewer blocks than the data file.

External Hashing

- Hashing with an external file is similar to internal hashing.

External Hashing

- Hashing with an external file is similar to internal hashing.
- In the **internal hashing** scheme, each element of the array table contains a pointer to the beginning of a chain of data items that hash into that location.

External Hashing

- Hashing with an external file is similar to internal hashing.
- In the **internal hashing** scheme, each element of the array table contains a pointer to the beginning of a chain of data items that hash into that location.
- In the **external hashing** scheme, each element of table still contains a pointer to the beginning of a chain, but here each chain consists of blocks of index records.
    - **You hash an index file rather than the data file.**

External Hashing

- A hashed index file

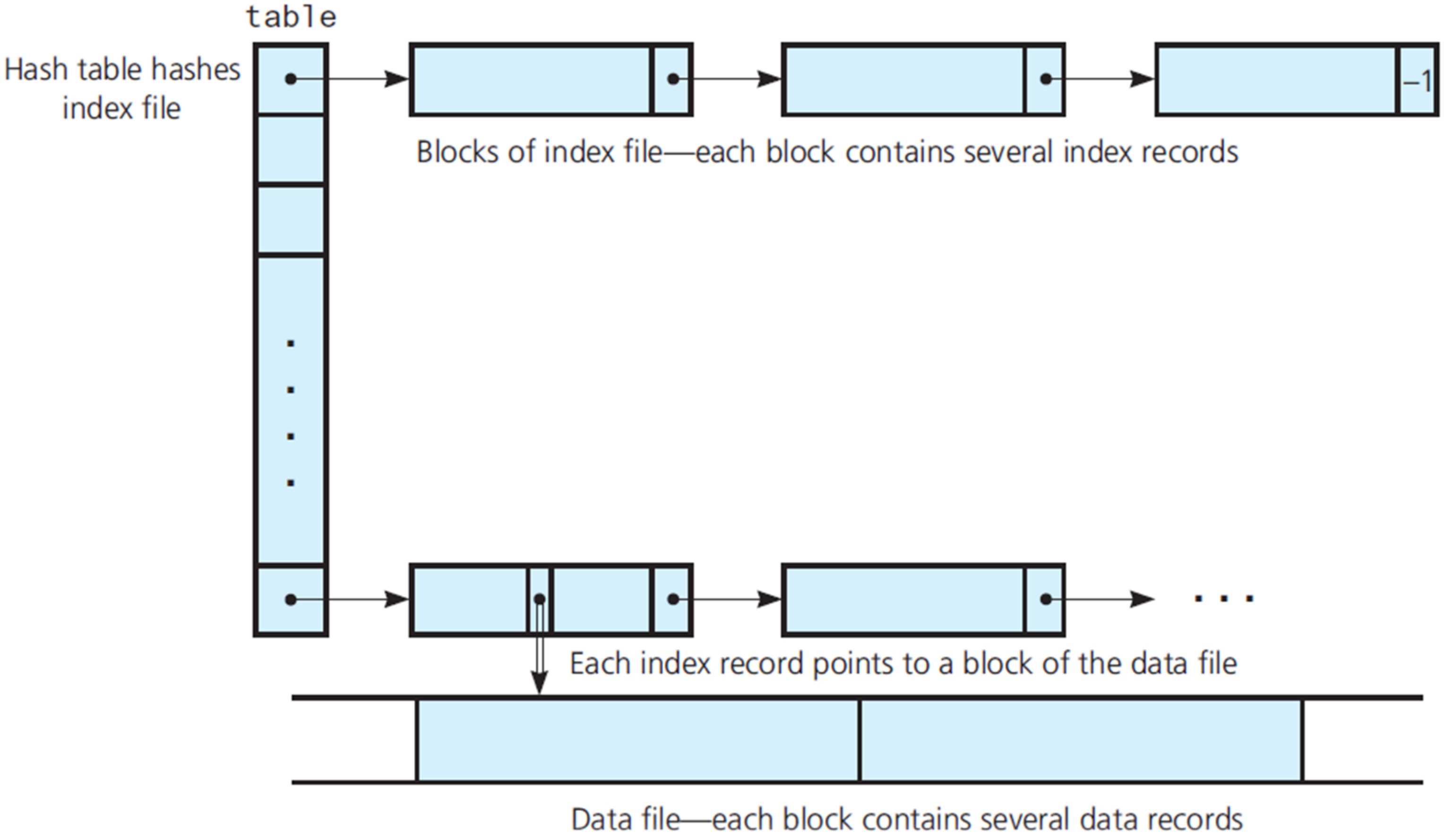


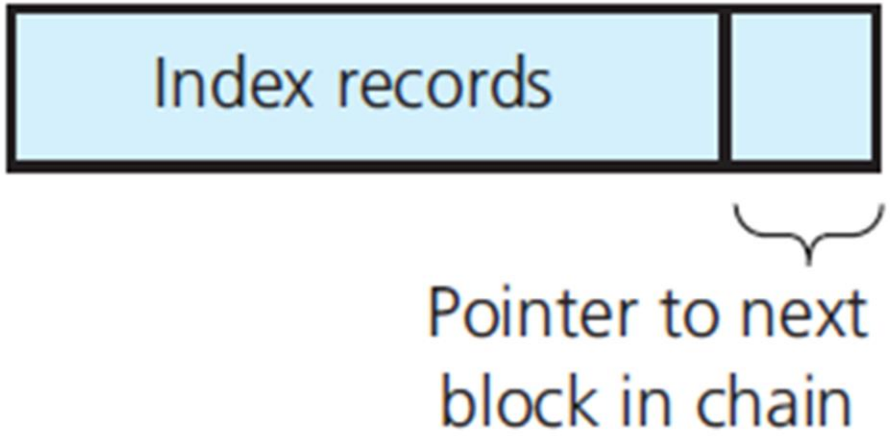
table
Hash table hashes index file
–1
Blocks of index file—each block contains several index records
. . .
Each index record points to a block of the data file
Data file—each block contains several data records

External Hashing

- Associated with each entry **table[i]** is a ***linked chain of blocks of the index file.***
- Each block of **table[i]**’s linked chain contains ***index records whose keys*** (and thus whose corresponding data records’ search keys) ***hash into location i***.

External Hashing

- Associated with each entry **table[i]** is a ***linked chain of blocks of the index file.***
- Each block of **table[i]**'s linked chain contains ***index records whose keys*** (and thus whose corresponding data records' search keys) ***hash into location i***.
- To form the linked chains, we must **reserve space** in each block for a block pointer – the integer block number of the next block in the chain.
- In this linked chain, the pointers are integers, not C++ pointers
- A single block with a pointer



External Hashing

- **Addition when an index file uses external hashing**
  1. Add data record into data file
  2. Add corresponding index record into index file

$$i = h(searchKey)$$

External Hashing

- **Removal when an index file uses external hashing**
  1. Search index file for corresponding index record

$$i = h\left(searchKey\right)$$

  2. Remove data record from data file

B-Trees

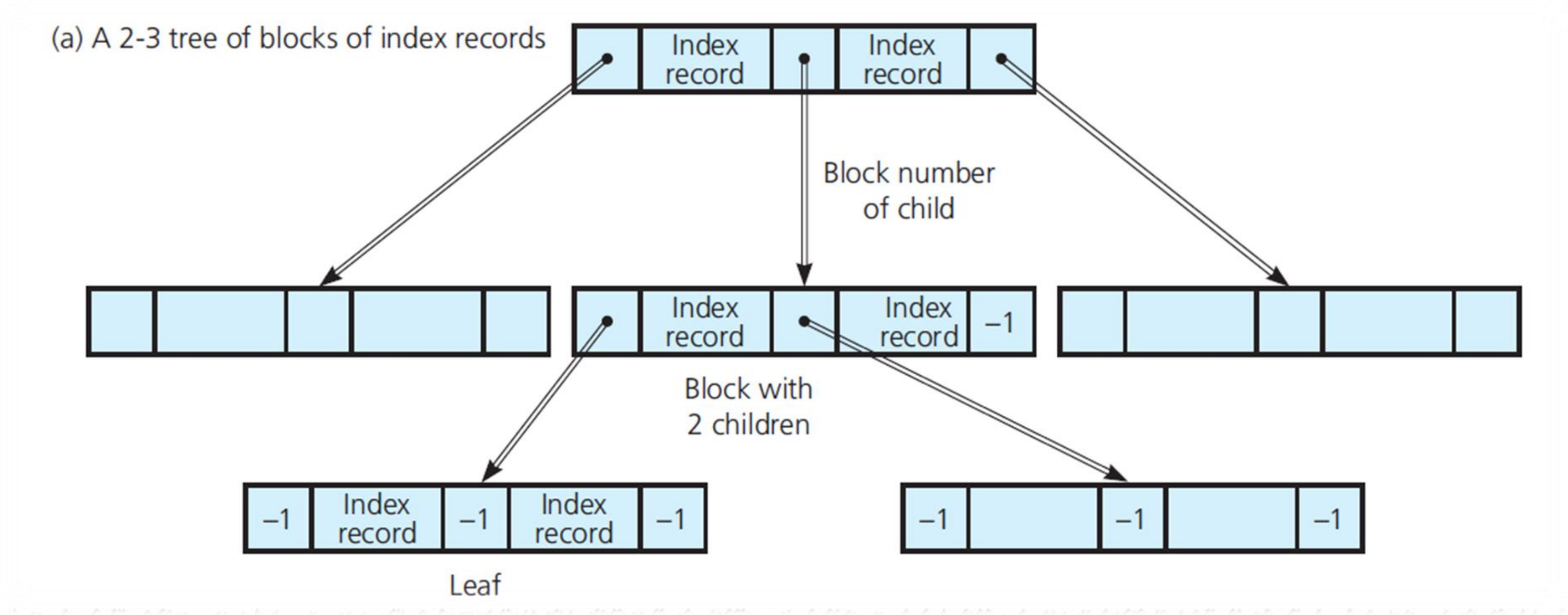
- Another way to search an external data collection is **to organize it as a Balanced Search Tree.**
- Concept is analogous to previous: just as you can apply external hashing to an index file, you can organize the index file, not the data file, as an external search tree.
- Use extensions of 2-3 trees.

B-Trees

- Organize blocks of an external file into a tree structure by using block numbers for child pointers.

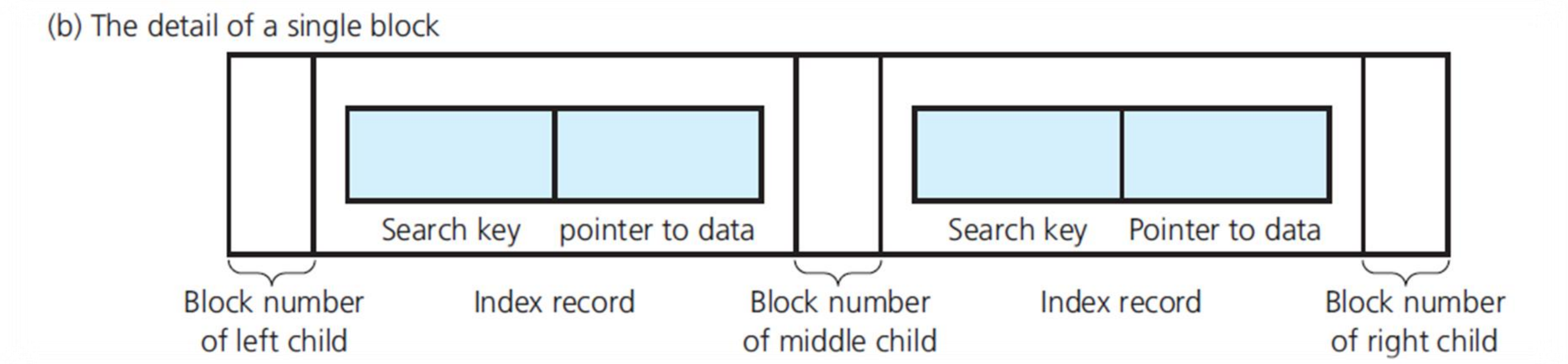
B-Trees

- Organize blocks of an external file into a tree structure by using block numbers for child pointers.
- **Blocks can be organized into a 2-3 tree, each block of the file being a node in the tree and contains three child pointers, each of which is the integer block number of the child.**



B-Trees

- An index file organizes as a 2-3 tree
- If you organized the index file into a 2-3 tree, each node (block of the index file) would contain either one or two index records, each of the form **<key, pointer>** and three child pointers.



(b) The detail of a single block

Search key
pointer to data
Search key
Pointer to data
Block number of left child
Index record
Block number of middle child
Index record
Block number of right child

- **The pointer portion of an index record has nothing to do with the tree structure of the index file; pointer indicates the block (in the data file) that contains the data record whose search key equals key.**

B-Trees

- Assuming index records in the tree are organized so that their keys obey the same search-tree ordering property as an internal 2-3 tree then retrieval takes the form:

```
// Searches the data file for the record whose search key equals searchKey.
// Returns the record if found, else throws NotFoundException
// rootNum is the block number (of the index file) that contains the root of the tree.
getItem(indexFile: File, dataFile: File, rootNum: integer, searchKey: KeyType): ItemType
{
        if (no blocks are left in the index file to read)
                throw NotFoundException
        else
        {
                // read from index file into internal array buf the block that contains the root of the 2-3 tree
                buf.readBlock(indexFile, rootNum)

                // Search for the index record whose key value equals searchKey
```

B-Trees

- Assuming index records in the tree are organized so that their keys obey the same search-tree ordering property as an internal 2-3 tree then retrieval takes the form:

```
                if (searchKey is in the root)
                {
                        blockNum = nmber of data-file block that index record specifies
                        data.readBlock(dataFile, blockNum)
                        Find data record data.getRecord(k) whose search key equals searchKey
                        return data.getRecord(k)
                }
                // Else search the appropriate subtree
                else if (the root is a leaf)
                        throw NotFoundException
                else
                {
                        Find the subtree Si to search
                        child = block number of the root of Si
                        return getItem(indexFile, dataFile, child, searchkey)
                }
        }
}
```

B-Trees

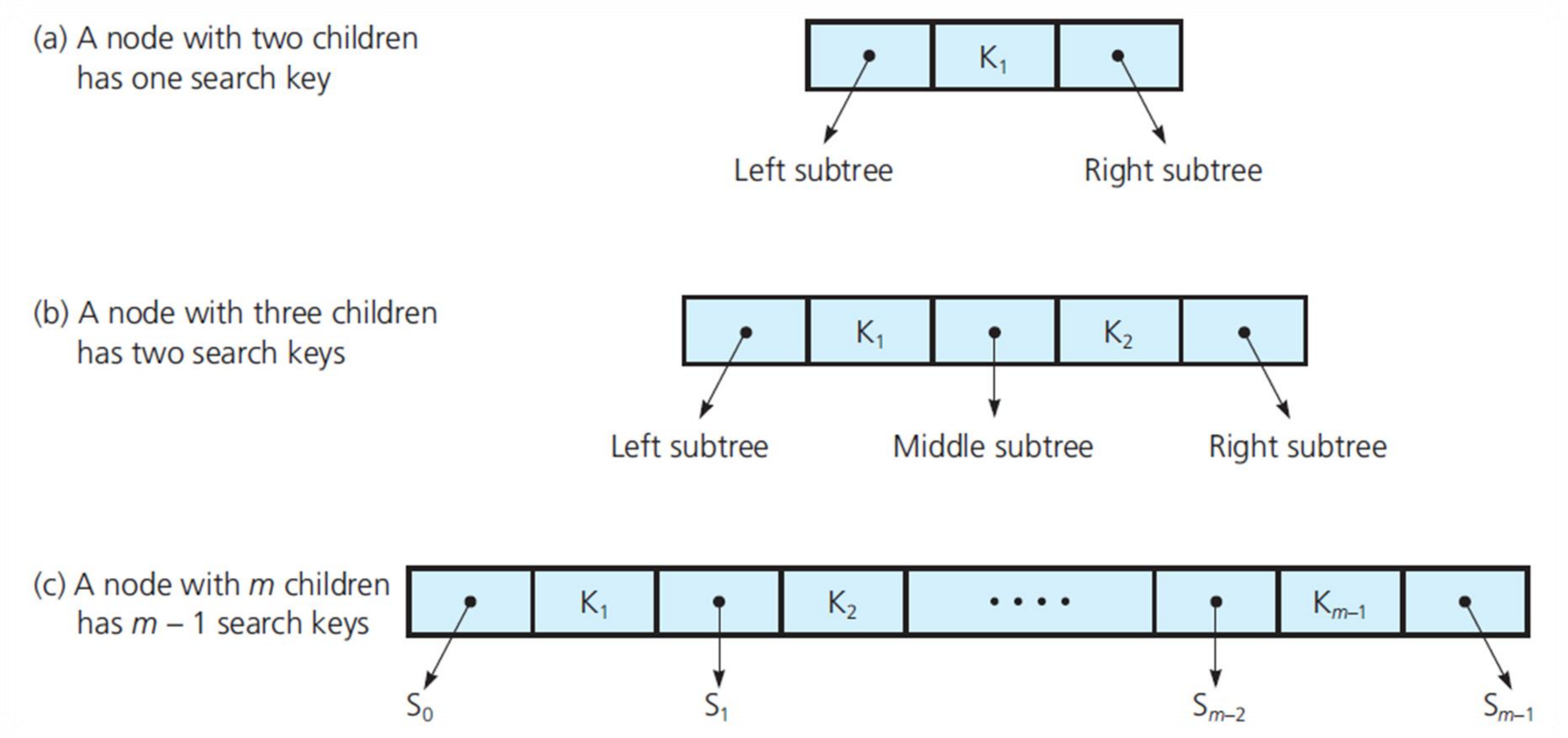
- How many children the nodes of the search tree can/should have? How big can m be?
- When we want to organize the index file into a search tree, the items that we store in each node will be records of the form **<key,pointer>**
- Thus, if each node in the tree is to have **m** children, it must be large enough to accommodate m child pointers and m-1 records of the form <key,pointer>
- **Choose m to be the largest integer such that m child pointers (which are integers) and m-1 <key,pointer> records can fit into a single block of the file.**

B-Trees

- **Number of children per node:** you should structure the external search tree so that every internal node has m children, where m is chosen as just described, and all leaves are the same level, as is the case with full trees and 2-3 trees.

B-Trees

- Nodes with two, three, and m children and their search keys



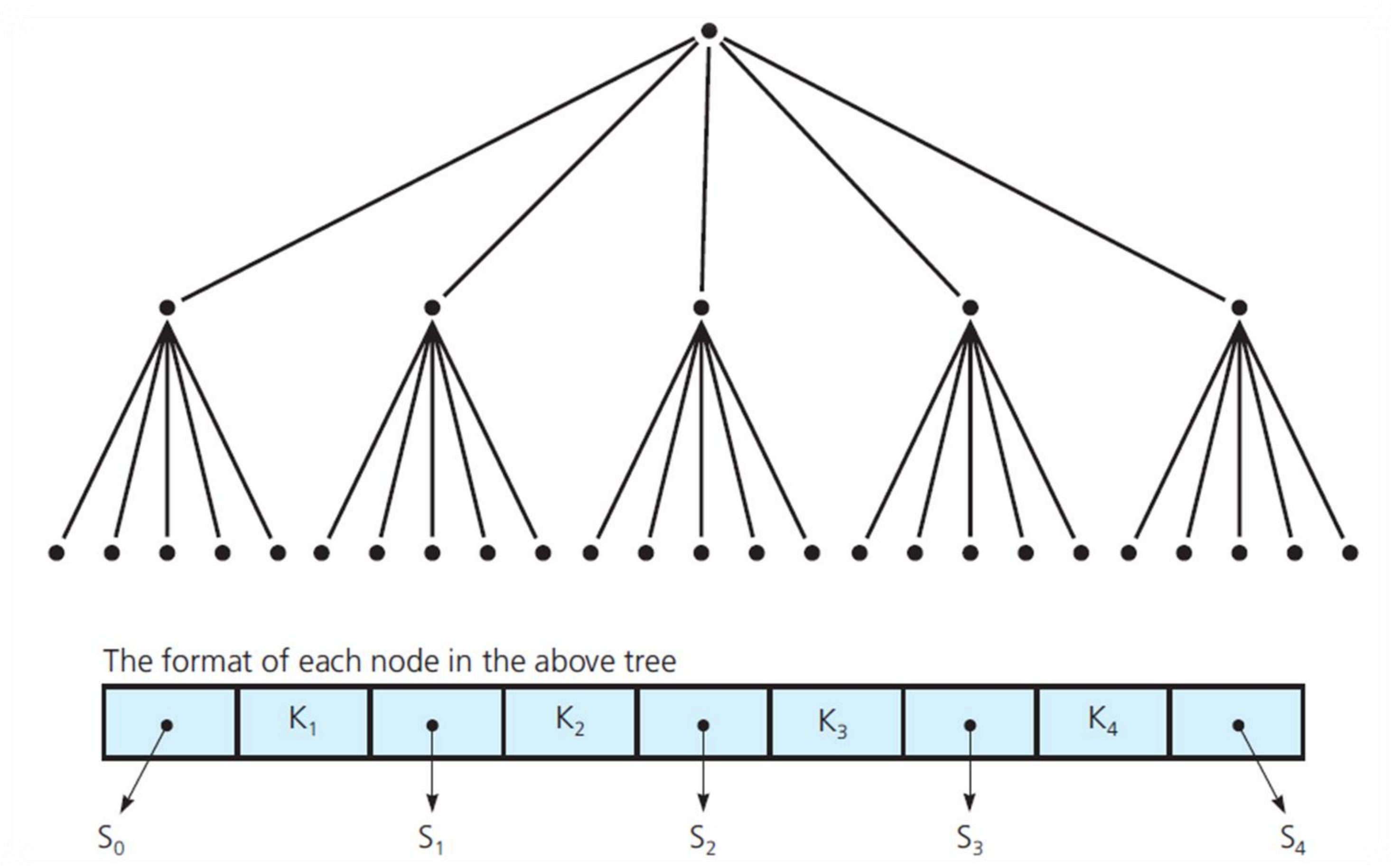
(a) A node with two children has one search key

(b) A node with three children has two search keys

(c) A node with *m* children has *m* – 1 search keys

B-Trees

- A full tree whose internal nodes have five children



The format of each node in the above tree

B-Trees

- Although this search tree has the minimum possible height, its balance is too difficult to maintain when adding or removing nodes.

B-Trees

- Although this search tree has the minimum possible height, its balance is too difficult to maintain when adding or removing nodes.
- **As a consequence, we need to make a compromise.**
  - We can either insist that all the leaves of the search tree be at the same level – that is, that the tree be balanced – but we must allow each internal node to have between m and [m/2]+1 children. ([]: greatest integer in.)
- **B-Tree of degree m**

B-Trees

- **B-Tree of degree m properties**
  - All leaves are at the same level
  - Each node contains between m-1 and [m/2] records, and each internal node has one more child than it has records.
    - Exception to this rule is that the root of the tree can contain as few as one record and can have as few as two children.
    - This exception is required by the addition and removal algorithms.

B-Trees

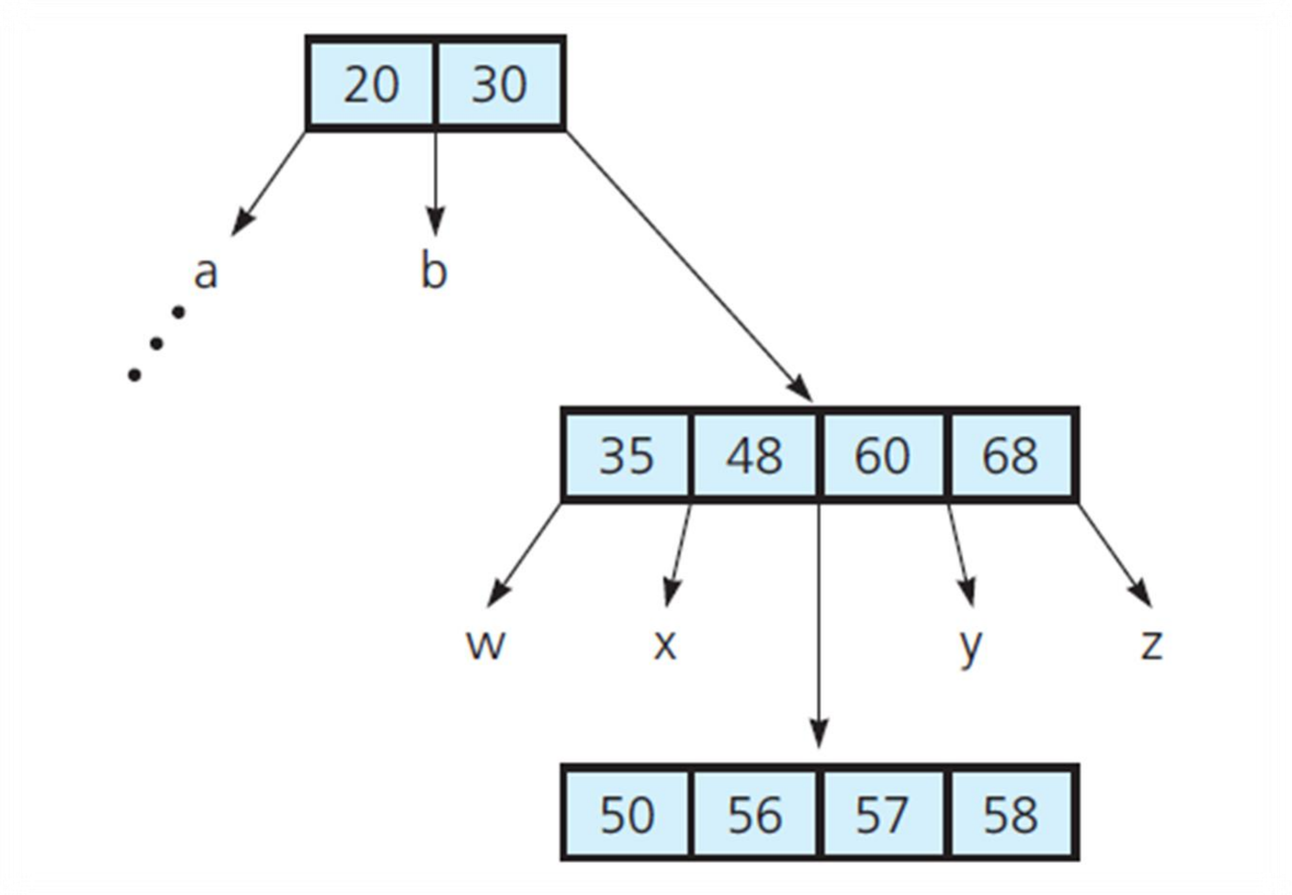
- **Adding a record to a B-tree**
  1. Add data record to data file
  2. Add a corresponding index record to index file

B-Trees

- **Adding a record to a B-tree**
  1. Add data record to data file:
     - First you find block p in the data file into which you can insert the new record.
     - As was true with the external hashing implementation, block p is either any block with a vacant slot or a new block.
  2. Add a corresponding index record to index file:
     - You now must add the index record <55,p> to the index file, which is a B-tree of degree 5.
     - The first step is to locate the leaf of the tree in which this index record belongs by determining where the search for 55 would terminate

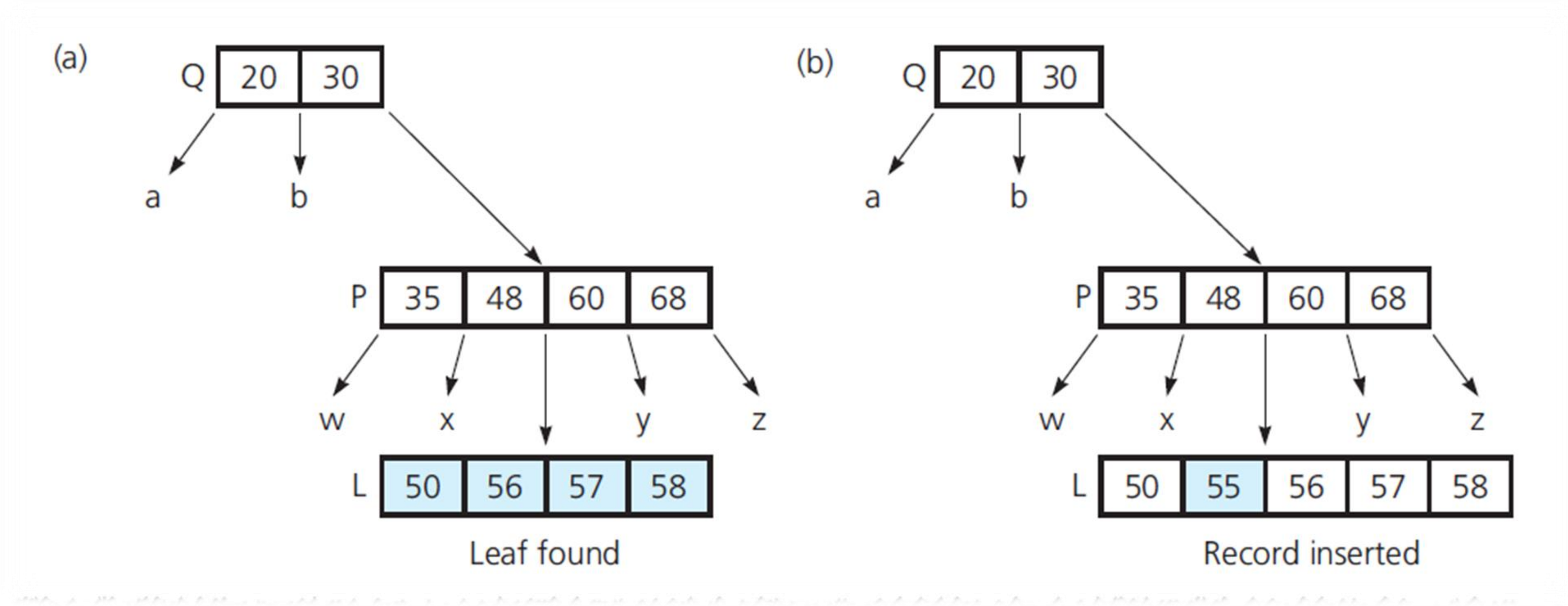
B-Trees

- A B-tree of degree 5



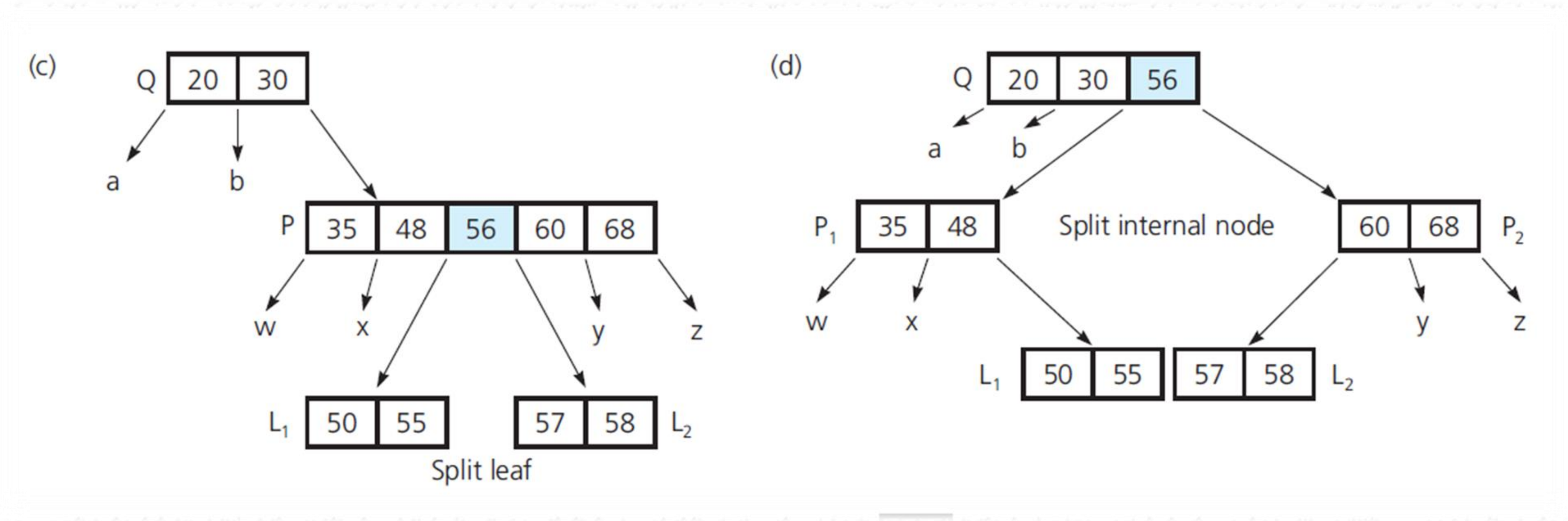
B-Trees

- The steps for adding 55 to a B-tree (a-d) and splitting the root (e)



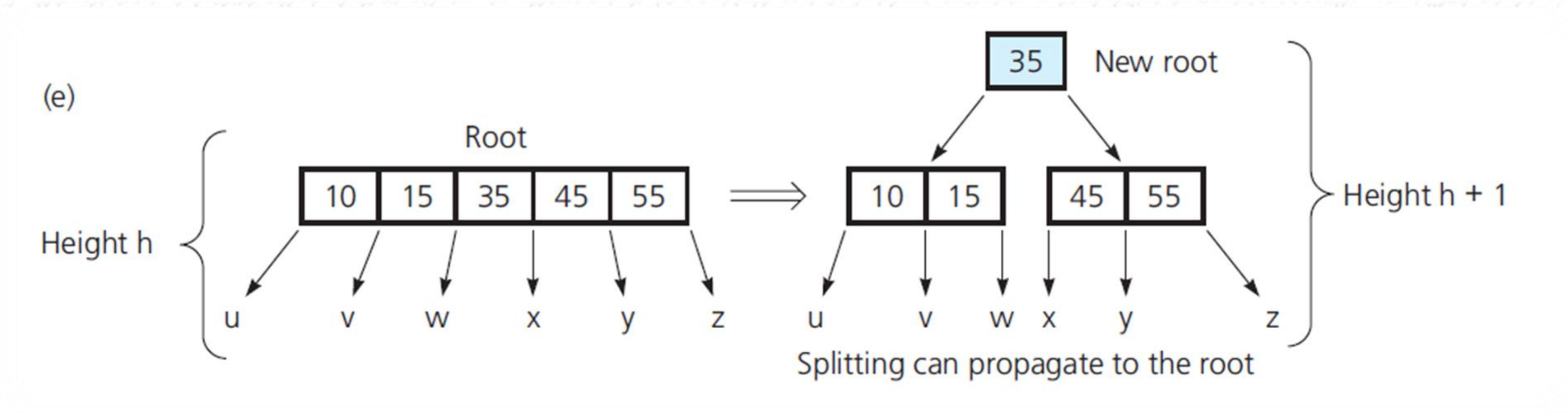
B-Trees

- The steps for adding 55 to a B-tree (a-d) and splitting the root (e)



B-Trees

- The steps for adding 55 to a B-tree (a-d) and splitting the root (e)



B-Trees

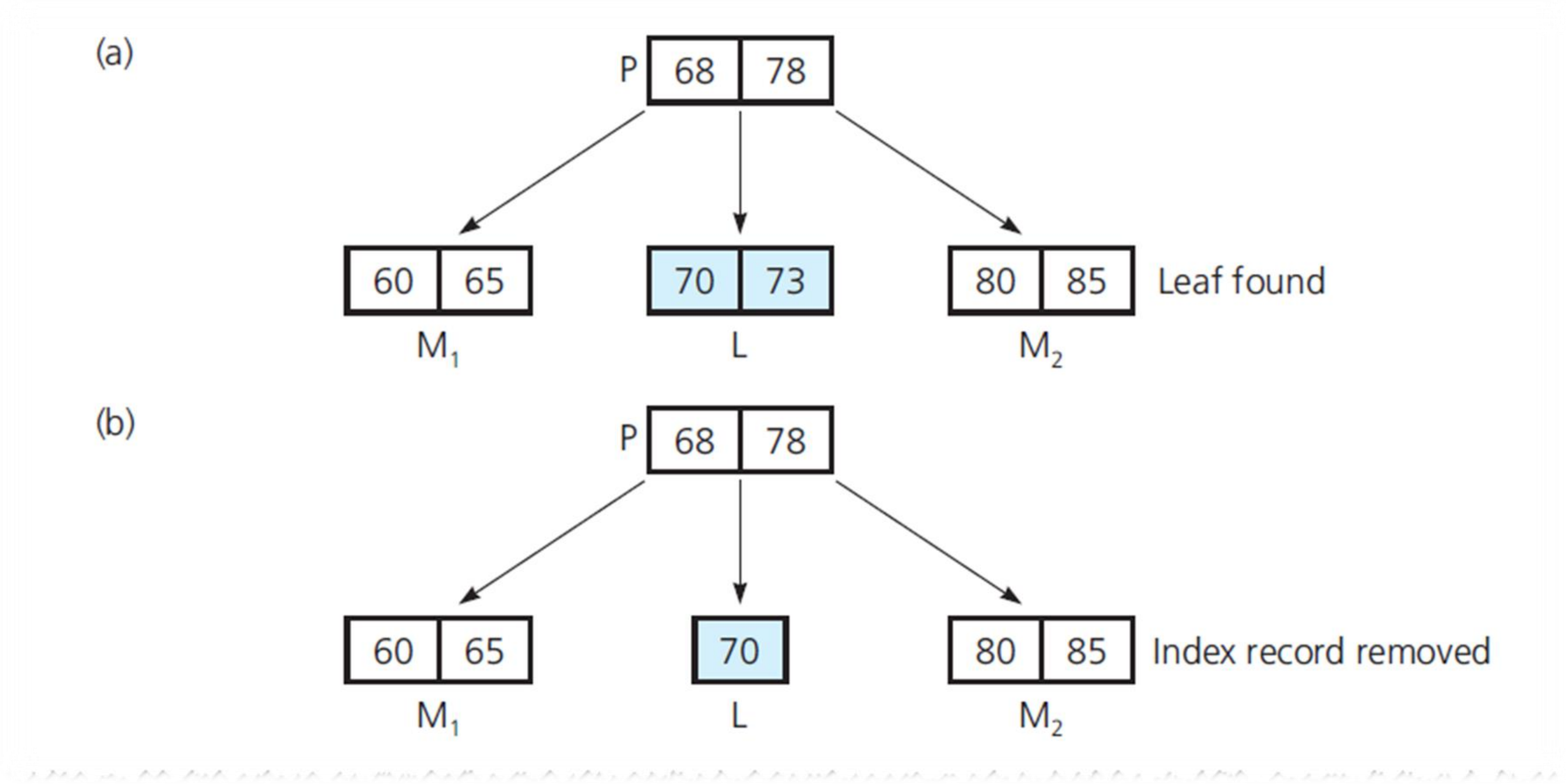
- **Removing a record from a B-tree**
    1. Locate index record in index file
    2. Remove data record from data file

B-Trees

- **Removing a record from a B-tree**
  1. Locate index record in index file
  2. Remove data record from data file

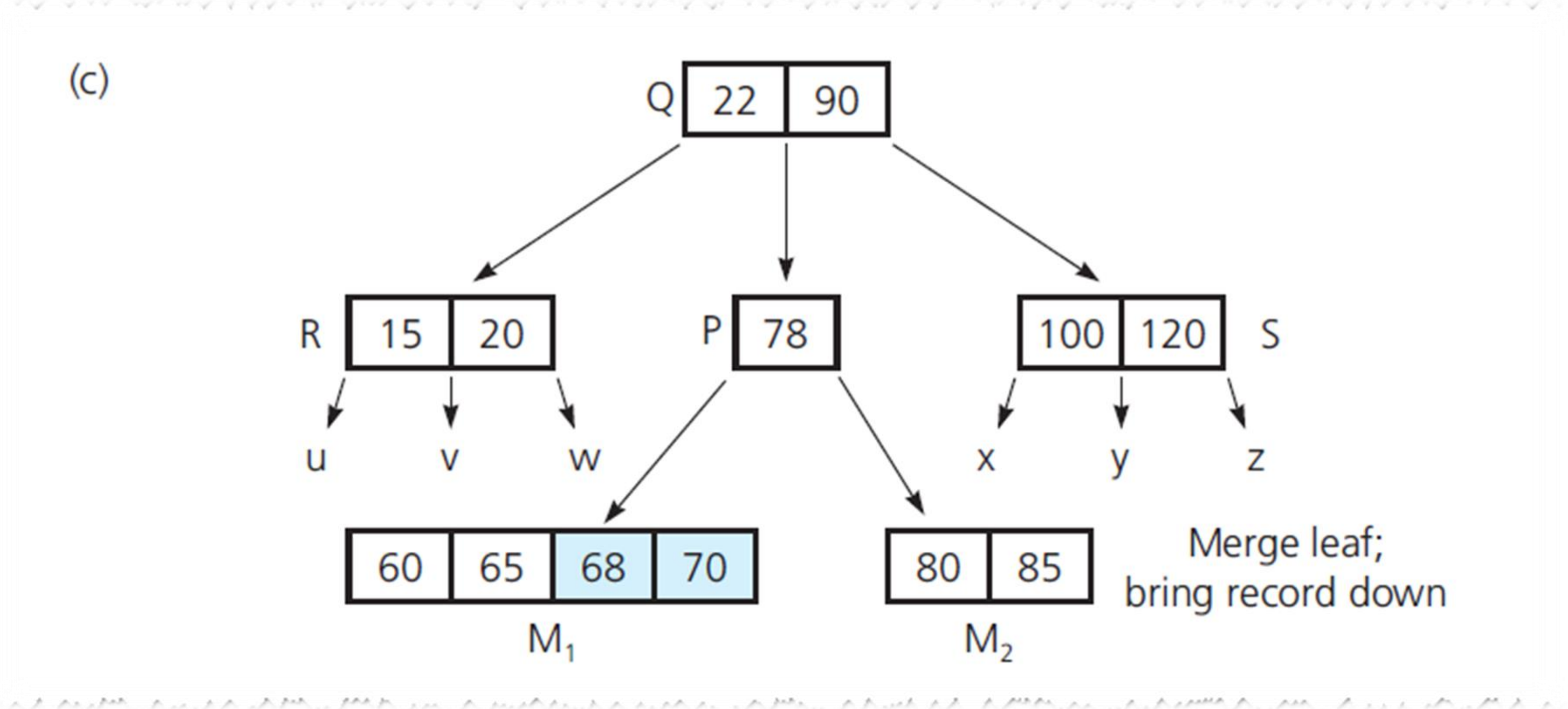
B-Trees

- The steps for removing 73 (a to e) and removing the root (f)



B-Trees

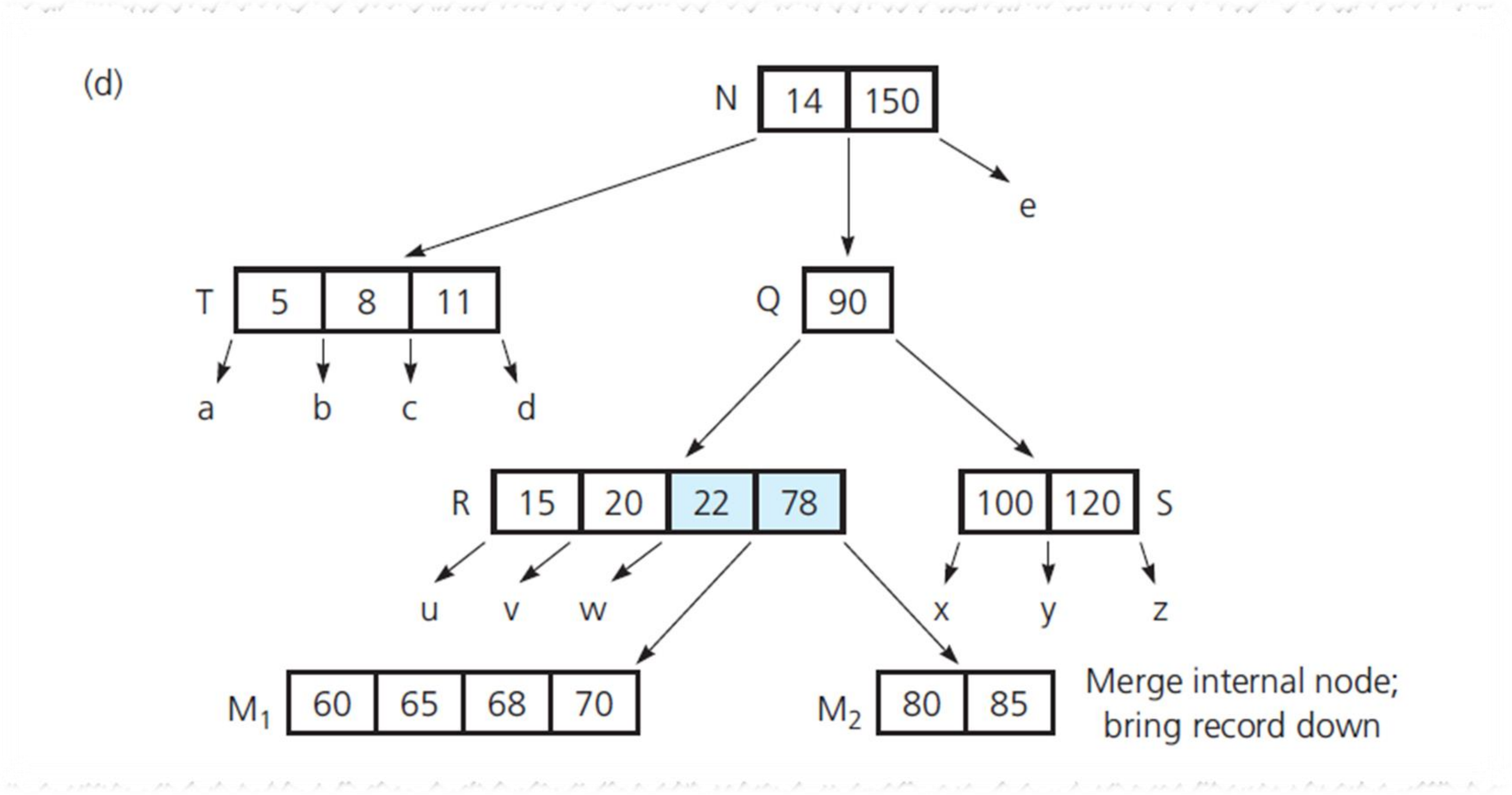
- The steps for removing 73 (a to e) and removing the root (f)



Merge leaf;
bring record down

B-Trees

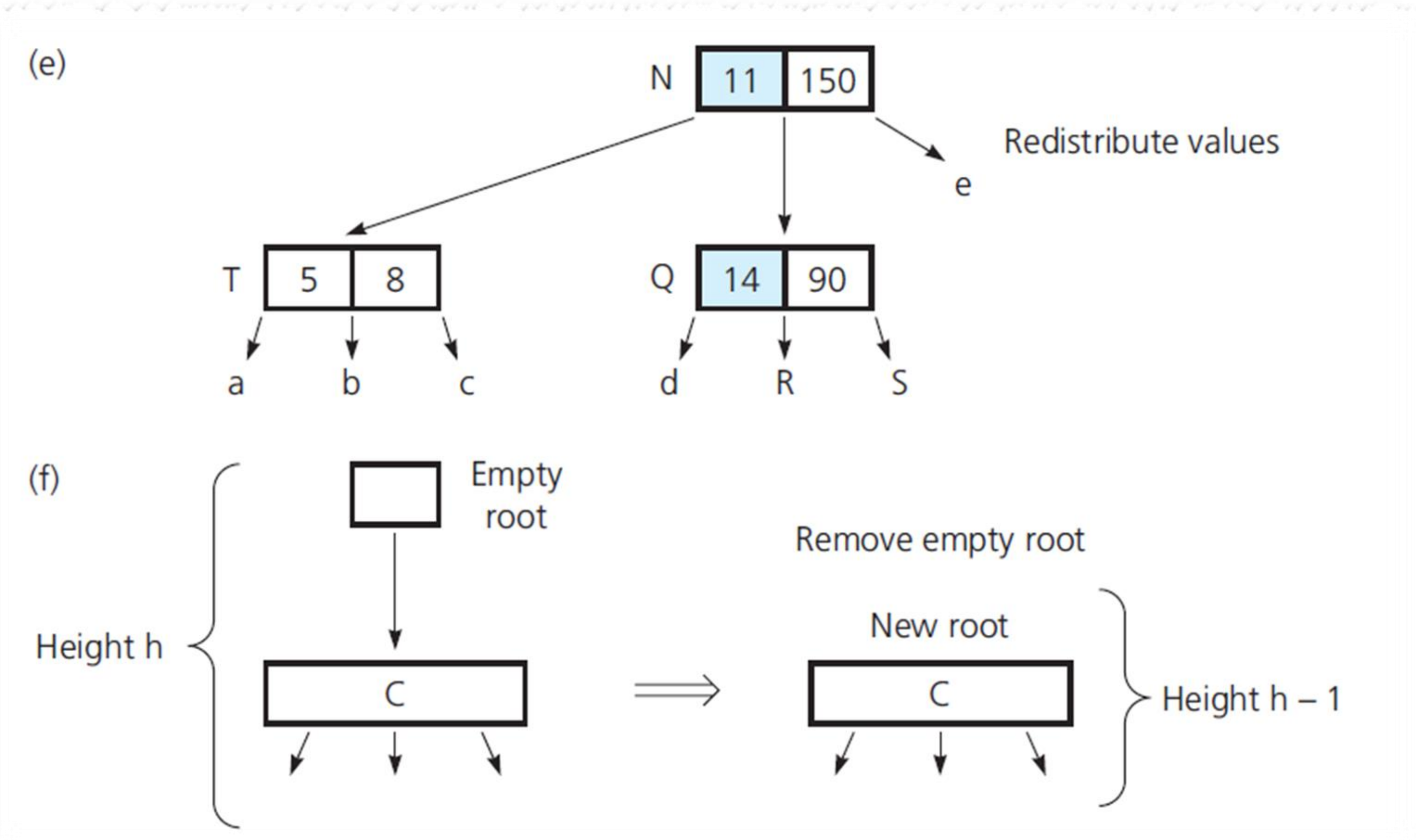
- The steps for removing 73 (a to e) and removing the root (f)



Merge internal node;
bring record down

B-Trees

- The steps for removing 73 (a to e) and removing the root (f)



B-Trees

- Inorder traversal of the B-tree

```
// Traverses in sorted order the search keys in an index file that is organized as a B-tree of degree m.
// blockNum is the block number of the root of the B-tree in the index file.
Traverse(blockNum: integer, m: integer): void
{
        if (blockNum != -1)
        {
                // Read the root into internal array buf
                buf.readBlock(indexFile, blockNum)

                // Traverse the children

                // Traverse S0
                Let p be the block number of the 0-th child of buf
                traverse(p, m)

                for (i = 1 through m-1)
                {
                        Display key Ki of buf

                        // Traverse Si
                        Let p be the block number of the i-th child of buf
                        traverse(p, m)
                }
        }
}
```

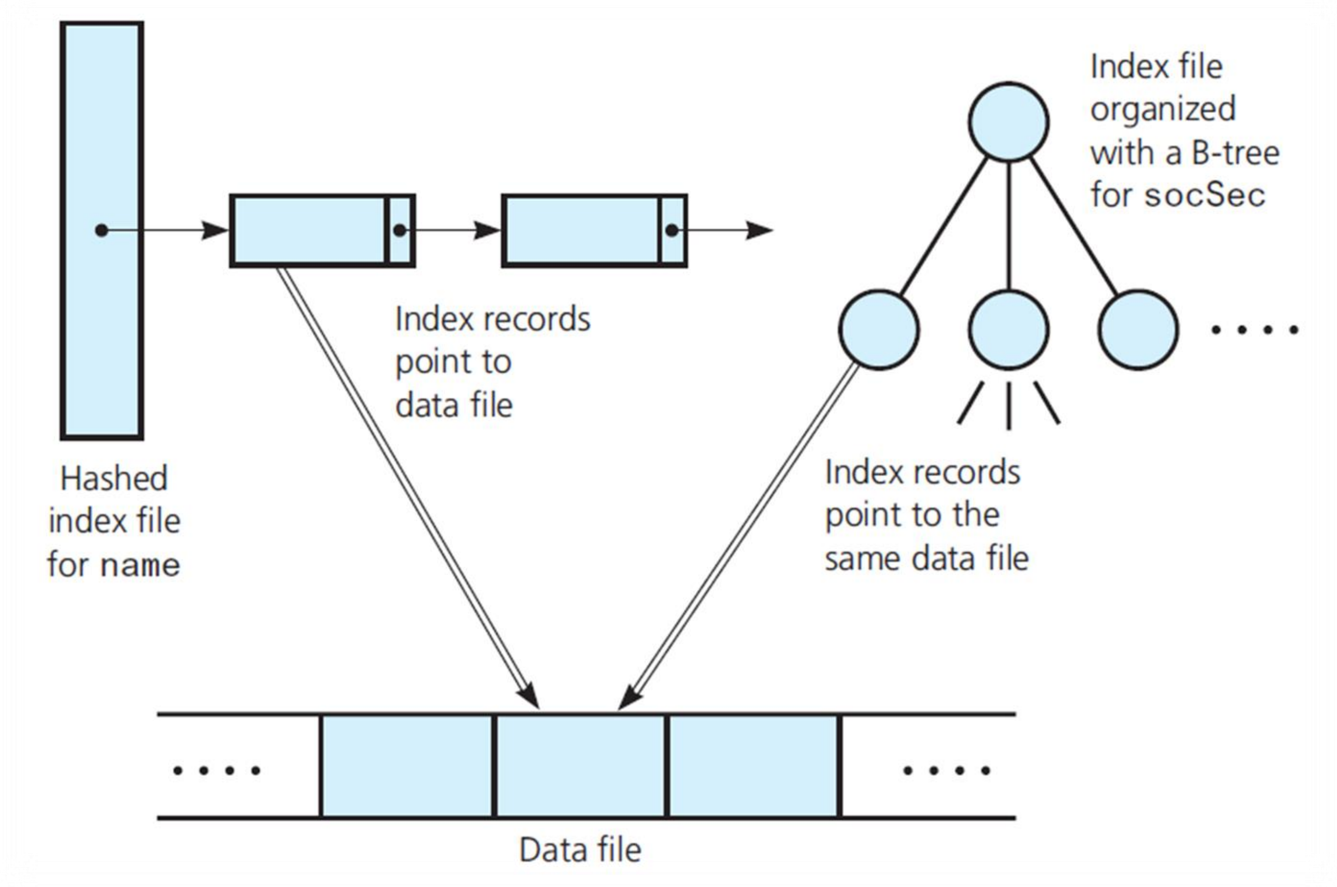
B-Trees

- Sorted order traversal of a datafile indexed with a B-tree

```
// Traverses in sorted order a data file that is indexed with a B-tree of degree m.
// blockNum is the block number of the root of the B-tree.
Traverse(blockNum: integer, m: integer): void
{
        if (blockNum != -1)
        {
                // Read the root into internal array buf
                buf.readBlock(indexFile, blockNum)

                // Traverse S0
                Let p be the block number of the 0-th child of buf
                traverse(p, m)

                for (i = 1 through m-1)
                {
                        Let p_i be the pointer in the i-th index record of buf
                        data.readBlock(dataFile, p_i)
                        Extract from data the data record whose search key equals Ki
                        Display the data reord

                        // Traverse Si
                        Let p be the block number of the i-th child of buf
                        traverse(p, m)
                }
        }
}
```

Multiple Indexing

- Multiple index files



Multiple Indexing

- A removal by name must update both indexes
    1. Search **name** index file for **jones** and remove index record.
    2. Remove appropriate data record from data file, noting **socSec** value **ssn** of this record.
    3. Search **socSec** index file for **ssn** and remove tis index record.

Thank you