CSI2131 FILE MANAGEMENT (PART II)

Prof. Lucia Moura

Winter 2003

LECTURE 15: HASHING I

Lecture 14 is skipped since it is a review lecture.

Contents of today's lecture:

- Introduction to Hashing
- Hash functions.
- Distribution of records among addresses, synonyms and collisions.
- Collision resolution by progressive overflow or linear probing.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 11.1,11.2,11.3,11.5.

Motivation

Hashing is a useful searching technique, which can be used for implementing indexes. The main motivation for Hashing is improving searching time.

Below we show how the search time for Hashing compares to the one for other methods:

- Simple Indexes (using binary search): $O(\log_2 N)$
- B Trees and B+ trees (will see later): $O(\log_k N)$
- Hashing: $O(1)$

What is Hashing ?

The idea is to discover the location of a key by simply examining the key. For that we need to design a hash function.

A **Hash Function** is a function $h(k)$ that transforms a key into an address.
An address space is chosen before hand. For example, we may decide the file will have 1,000 available addresses.

If U is the set of all possible keys, the hash function is from U to {0,1,...,999}, that is

$$h : U \longrightarrow \{0, 1, ..., 999\}$$

Example :

k	**ASCII code for the first 2 letters**	`product`	**h(k) = `product` mod 1,000**
BALL	66, 65	$66 \times 65 = 4,290$	290
LOWELL	76, 79	$76 \times 79 = 6,004$	004
TREE	84, 82	$84 \times 82 = 6,888$	888

RRN	FILE
000	
001	
⋮	⋮
004	LOWELL
⋮	⋮
290	BALL
⋮	⋮
888	TREE
⋮	⋮
999	

There is no obvious connection between the key and the location (randomizing).

Two different keys may be sent to the same address generating a **Collision**.

Can you give an example of collision for the hash function in the previous example ?

Answer:
LOWELL, LOCK, OLIVER, and any word with first two letters L and O will be mapped to the same address:

$$h(\text{LOWELL}) = h(\text{LOCK}) = h(\text{OLIVER}) = 4.$$

These keys are called **synonyms**. The address “4” is said to be the **home address** of any of these keys.

Avoiding collisions is extremely difficult **(do you know the birthday paradox?)**, so we need techniques for dealing with it.

Ways of reducing collisions:

1. **Spread out the records** by choosing a good hash function.
2. **Use extra memory**, i.e. increase the size of the address space (Ex: reserve 5,000 available addresses rather than 1,000).
3. **Put more than one record at a single address** (use of buckets).

A Simple Hash Function

To compute this hash function, apply 3 steps :

Step 1: transform the key into a number.

```
LOWELL  =  | L | O | W | E | L | L |  |  |  |  |  |  |
ASCII code: 76  79  87  69  76  76  32 32 32 32 32 32
```

Step 2: fold and add (chop off pieces of the number and add them together) and take the mod by a prime number

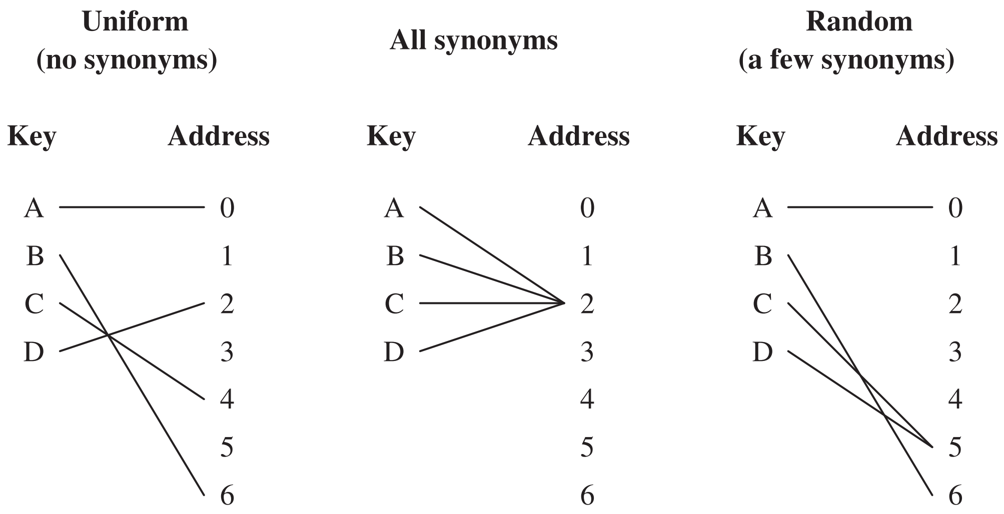
```
7679|8769|7676|3232|3232|3232|

7679+8769+7676+3232+3232+3232 = 33,820

33,820 mod 19937 = 13,883
```

Step 3: divide by the size of the address space (preferably a prime number).

```
13,883 mod 101 = 46
```

Distribution of Records among Addresses

There are 3 possibilities :



Uniform distributions are extremely rare.

Random distributions are acceptable and more easily obtainable.

Trying a **better-than-random** distribution, by preserving natural ordering among the keys :

- Examine keys for patterns.

  Ex: Numerical keys that are spread out naturally such as:
  keys are years between 1970 and 2000.
  $f(year) = (year - 1970) \bmod (2000 - 1970 + 1)$
  $f(1970) = 0, f(1971) = 1, \cdots, f(2000) = 30$

- Fold parts of the key.

  Folding means extracting digits from a key and adding the parts together as in the previous example.
  In some cases, this process may preserve the natural separation of keys, if there is a natural separation.

- Use prime number when dividing the key.

  Dividing by a number is good when there are sequences of consecutive numbers.

  If there are many different sequences of consecutive numbers, dividing by a number that has many small factors may result in lots of collisions. A prime number is a better choice.

When there is no natural separation between keys, try **randomization**.

You can using the following Hash functions:

- **Square the key and take the middle**:

  Ex: key = 453 $\quad 453^2 = 205209$
  Extract the middle = 52.
  This address is between 00 and 99.

- **Radix transformation**:

  Transform the number into another base and then divide by the maximum address.

  Ex: Addresses from 0 to 99
  key = 453 in base 11 : 382
  hash address = 382 mod 99 = 85.

Collision Resolution:Progressive Overflow

Progressive overflow/linear probing works as follows :

Insertion of key k:

- Go to the home address of k : h(k)
- If free, place the key there
- If occupied, try the next position until an empty position is found (the 'next' position for the last position is position 0, i.e. wrap around)

Example :

key k	Home address - h(k)
COLE	20
BATES	21
ADAMS	21
DEAN	22
EVANS	20

Complete Table:

0	
1	
2	
⋮	⋮
19	
20	
21	
22	

Table size = 23

Searching for key k:

- Go to the home address of k : h(k)
- If k is in home address, we are done.
- Otherwise try the next position until: key is found or empty space is found or home address is reached (in the last 2 cases, the key is not found)

0	DEAN
1	EVANS
2	
⋮	⋮
19	
20	COLE
21	BATES
22	ADAMS

Ex :
A search for 'EVANS' probes places : 20, 21, 22, 0, 1, finding the record at position 1.

Search for 'MOURA', if h(MOURA)=22, probes places 22, 0, 1, 2 where it concludes 'MOURA' in not in the table.

Search for 'SMITH', if h(SMITH)=19, probes 19, and concludes 'SMITH' in not in the table.

Advantage : Simplicity

Disadvantage : If there are lots of collisions, clusters of records can form, as in the previous example.

Search length

- Number of accesses required to retrieve a record.

```
average search length
= (sum of search lengths)/(numb.of records)
```

In the previous example :

0	DEAN
1	EVANS
2	
⋮	⋮
19	
20	COLE
21	BATES
22	ADAMS

key	**Search Length**
COLE	
BATES	
ADAMS	
DEAN	
EVANS	

Average search length = (1+1+2+2+5)/5 = 2.2.

Refer to figure 11.7 in page 489. It shows that a packing density up to 60% gives an average search length of 2 probes, but higher packing densities make search length to increase rapidly.

LECTURE 16: HASHING II

Contents of today's lecture:

- Predicting record distribution; packing density.
- Hashing with Buckets
- Implementation issues.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 11.3,11.4,11.6

Predicting Record Distribution

Throughout this section we assume a random distribution for the hash function.

Let N = number of available addresses, and
r = number of records to be stored.

Let $p(x)$ be the probability that a given address will have x records assigned to it.

It is easy to see that

$$p(x) = \frac{r!}{(r-x)!x!} \left[1 - \frac{1}{N}\right]^{r-x} \left[\frac{1}{N}\right]^{x}$$

For N and r large enough this can be approximated by:

$$p(x) \sim \frac{(r/N)^x \, e^{-(r/N)}}{x!}$$

Example : $N = 1{,}000$, $r = 1{,}000$

$$p(0) \sim \frac{1^0\, e^{-1}}{0!} = 0.368$$

$$p(1) \sim \frac{1^1\, e^{-1}}{1!} = 0.368$$

$$p(2) \sim \frac{1^2\, e^{-1}}{2!} = 0.184$$

$$p(3) \sim \frac{1^3\, e^{-1}}{3!} = 0.061$$

For N addresses,
the expected number of addresses with x records is

$$N \cdot p(x).$$

Complete the numbers below for the example above:

expected # of addresses with 0 records assigned to it =
expected # of addresses with 1 records assigned to it =
expected # of addresses with 2 records assigned to it =
expected # of addresses with 3 records assigned to it =

Reducing Collision by using more Addresses

Now, we see how to reduce collisions by increasing the number of available addresses.

DEFINITION: **`packing density`** $= r/N$

500 records to be spread over 1000 addresses result in **`packing density`** $= 500/1000 = 0.5 = 50\%$.

Some questions :

1. How many addresses go unused ? *More precisely: What is the* **expected** *number of addresses with no key mapped to it?*

   $N \cdot p(0) = 1000 \cdot 0.607 = 607$

2. How many addresses have no synonyms ? *More precisely: What is the expected number of address with only one key mapped to it?*

   $N \cdot p(1) = 1000 \cdot 0.303 = 303$

3. How many addresses contain 2 or more synonyms ? *More precisely: What is the expected number of addresses with two or more keys mapped to it ?*

   $N \cdot (p(2) + p(3) + ...) = N \cdot (1 - (p(0) + p(1)) = 1000 \cdot 0.09 = 90$

4. Assuming that only one record can be assigned to an address, how many overflow records are expected ?

   $1 \cdot N \cdot p(2) + 2 \cdot N \cdot p(3) + 3 \cdot N \cdot p(4) + ... =$
   $N \cdot [p(2) + 2 \cdot p(3) + 3 \cdot p(4) + ...] \sim 107.$

   The justification for the above formula is that there is going to be $(i - 1)$ overflow records for all the table positions that have i records mapped to it, which are expected to be as many as $N \cdot p(i)$.
   Now, there is a simpler formula derived by students from 2001:

   ```
   expected # of overflow records =
   = (#records) - (expected # of nonoverflow records)
   ```

   $= r - (N \cdot p(1) + N \cdot p(2) + N \cdot p(3) + \ldots)$
   $= r - N \cdot (1 - p(0)) \quad$ (since probabilities add up to 1)
   $= N \cdot p(0) - (N - r)$

   ```
   =(expect. # of empty posit. for random hash function)
     - (# of empty positions for perfect hash function)
   ```

   Using this formula we get the same result as before:
   $N \cdot p(0) - (N - r) = 607 - 500 = 107$

5. What is the expected percentage of overflow records ?
   $107/500 = 0.214 = 21.4\%$

Note that using either formula, the percentage of overflow records depend only on the packing density ($PD = r/N$), and not on the individual values of N or r.
Indeed, using the formulas derived in 4., we get that the percentage of overflow records is:

$$\frac{r - N \cdot (1 - p(0))}{r} = 1 - \frac{1}{PD} \cdot (1 - p(0))$$

and the Poisson function that approximate $p(0)$ is a function of r/N which is equal to PD (for hashing without buckets).
So, hashing with packing density $PD = 50\%$ always yield 21% of records stored outside their home addresses.
Thus, we can compute the expected percentage of overflow records, given the packing density:

packing density %	% **overflow records**
10%	4.8%
20%	9.4%
30%	13.6%
40%	17.6%
50%	21.4%
60%	24.8%
70%	28.1%
80%	31.2%
90%	34.1%
100%	36.8%

Hashing with Buckets

This is a variation of hashed files in which more than one record/key is stored per hash address.

bucket = block of records corresponding to one address in the hash table.

The hash function gives the **Bucket Address**.

Example: for a bucket holding 3 records, insert the following keys:

key	**Home Address**
LOYD	34
KING	33
LAND	33
MARX	33
NUTT	33
PLUM	34
REES	34

0	
⋮	⋮
33	
34	

Effects of Buckets on Performance

We should slightly change some formulas:

$$\texttt{packing density} = \frac{r}{b \cdot N}$$

We will compare the following two alternatives:

1. Storing 750 data records into a hashed file with 1,000 addresses, each holding 1 record.

2. Storing 750 data records into a hashed file with 500 bucket addresses, each bucket holding 2 records.

- In both cases the packing density is 0.75 or 75%.
- In the first case r/N=0.75.
  In the second case r/N=1.50.

Estimating the probabilities as defined before:

	p(0)	**p(1)**	**p(2)**	**p(3)**	**p(4)**
1) r/N=0.75 (b=1)	0.472	0.354	0.133	0.033	0.006
2) r/N=1.50 (b=2)	0.223	0.335	0.251	0.126	0.047

Calculating the number of overflow records in each case:

1. **b=1** (r/N=0.75):
   Number of overflow records =
   $= N \cdot [1 \cdot p(2) + 2 \cdot p(3) + 3 \cdot p(4) + \ldots]$
   $= r - N\,[p(1) + p(2) + p(3) + \ldots]$ (formula derived last class)
   $= r - N \cdot (1 - p(0))$
   $= 750 - 1000 \cdot (1 - 0.472) = 750 - 528 = 222.$
   This is about 29.6% overflow.

2. **b=2** (r/N=1.5):
   Number of overflow records =
   $= N \cdot [1 \cdot p(3) + 2 \cdot p(4) + 3 \cdot p(5) + \ldots]$
   $= r - N \cdot p(1) - 2 \cdot N \cdot [p(2) + p(3) + \ldots]$ (formula for b=2)
   $= r - N \cdot [p(1) + 2[1 - p(0) - p(1)]$
   $= r - N \cdot [2 - 2 \cdot p(0) - p(1)]$
   $= 750 - 500 \cdot [2 - 2 \cdot (0.223) - 0.335] = 140.5 \cong 140.$
   This is about 18.7% overflow.

Indeed, the percentage of collisions for different bucket sizes is:

	Bucket Size				
Packing Density %	**1**	**2**	**5**	**10**	**100**
75%	29.6%	18.7%	8.6%	4.0%	0.0%

Refer to table 11.4 page 495 of the book to see the percentage of collisions for different packing densities and different bucket sizes.

Implementation Issues:

1. Bucket structure

   A Bucket should contain a counter that keeps track of the number of records stored in it. Empty slots in a bucket may be marked '//.../'.
   Ex: Bucket of size 3 holding 2 records:

2	JONES	/////////...///	ARNSWORTH

2. Initializing a file for hashing:

   - Decide on the **Logical Size** (number of available addresses) and on the number of buckets per address.
   - Create a file of empty buckets before storing records. An empty bucket will look like:

0	/////////...///	/////////...///	/////////...///

3. Loading a hash file:

   When inserting a key, remember to:
   - Wrap around when searching for available bucket.
   - Be careful with infinite loops when hash file is full.

LECTURE 17: HASHING III

Contents of today's lecture:

- Deletions in hashed files.
- Other collision resolution techniques:
  - double hashing,
  - chained progressive overflow,
  - chaining with separate overflow area,
  - scatter tables.
- Patterns of record access.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 11.7,11.8,11.9

Making Deletions

Deletions in a hashed file have to be made with care.

Record	ADAMS	JONES	MORRIS	SMITH
Home Address	5	6	6	5

Hashed File using Progressive Overflow:

⋮	⋮
4	//////////
5	ADAMS
6	JONES
7	MORRIS
8	SMITH
⋮	⋮

Delete 'MORRIS'

If 'MORRIS' is simply erased, a search for 'SMITH' would be unsuccessful:

⋮	⋮	
4	//////////	← empty slot
5	ADAMS	
6	JONES	
7	//////////	← empty slot (WRONG: can't find 'SMITH' !!!)
8	SMITH	
⋮	⋮	

Search for 'SMITH' would go to home address (position 5) and when reached 7 it would conclude 'SMITH' is not in the file!

IDEA: use **TOMBSTONES**, i.e. replace deleted records with a marker indicating that a record once lived there:

⋮	⋮	
4	//////////	
5	ADAMS	
6	JONES	
7	#######	← tombstone (CORRECT: will find 'SMITH')
8	SMITH	
⋮	⋮	

A search must continue when it finds a tombstone, but can stop whenever an empty slot is found. A search for 'SMITH' will continue when if finds the tombstone in position 7 of the above table.

Note: Only insert a **tombstone** when the next record is occupied or is a tombstone. If the next record is an empty slot, we may mark the deleted record as empty. Why ?

Insertions should be modified to work with **tombstones**: if either an empty slot or a tombstone is reached, place the new record there.

Effects of Deletions and Additions on Performance

The presence of too many tombstones increases search length.

Solutions to the problem of deteriorating average search lengths:

1. Deletion algorithm may try to move records that follow a tombstone backwards towards its home address.
2. Complete reorganization: re-hashing.
3. Use a different type of collision resolution technique.

Other Collision Resolution Techniques

1) Double Hashing

- The first hash function determines the home address.
- If the home address is occupied, apply a second hash function to get a number c (c relatively prime to N).
- c is added to the home address to produce an overflow addresses; if occupied, proceed by adding c to the overflow address, until an empty spot is found.

Example:

k (key)	ADAMS	JONES	MORRIS	SMITH
$h_1(k)$ (home address)	5	6	6	5
$h_2(k) = c$	2	3	4	3

Hashed file using double hashing:

0	
1	
2	
3	
4	
5	ADAMS
6	JONES
7	
8	SMITH
9	
10	MORRIS

0	XXXXX
1	XXXXX
2	XXXXX
3	XXXXX
4	XXXXX
5	XXXXX
6	XXXXX
7	XXXXX
8	XXXXX
9	XXXXX
10	XXXXX

Suppose the above table is full, and that a key $\overline{k}$ has $h_1(\overline{k}) = 6$ and $h_2(\overline{k}) = 3$.

Question: What would be the order in which the addresses would be probed when trying to insert $\overline{k}$?

Answer: 6, 9, 1, 4, 7, 10, 2, 5, 8, 0, 3.

2) Chained Progressive Overflow

- Similar to progressive overflow, except that synonyms are linked together with pointers.
- The objective is to reduce the search length for records within clusters.

Example X: Search lengths:

Key	**Home**	**Progressive Overflow**	**Chained Progr. Overflow**
ADAMS	20	1	1
BATES	21	1	1
COLES	20	3	2
DEAN	21	3	2
EVANS	24	1	1
FLINT	20	6	3
Average Search Length :		2.5	1.7

Progressive Overflow

	data
⋮	⋮
20	ADAMS
21	BATES
22	COLES
23	DEAN
24	EVANS
25	FLINT
⋮	⋮

Chained Progressive Overflow

	data	**next**
⋮	⋮	⋮
20	ADAMS	22
21	BATES	23
22	COLES	25
23	DEAN	-1
24	EVANS	-1
25	FLINT	-1
⋮	⋮	⋮

PROBLEM: Suppose that 'DEAN' home address is 22. Since 'COLES' is there, we couldn't have a link to 'DEAN' starting in its home address!

Solution:
Two-pass loading:
- First pass: only load records that fit into their home addresses.
- Second pass: load all overflow records.

Care should be taken when deletions are done.

key	**home address**
ADAMS	20
BATES	21
COLES	20
DEAN	22
EVANS	24
FLINT	20

table after first pass:

20	ADAMS	-1
21	BATES	-1
22	DEAN	-1
23		
24	EVANS	-1
25		

table after second pass:

20	ADAMS	23
21	BATES	-1
22	DEAN	-1
23	COLES	25
24	EVANS	-1
25	FLINT	-1

3) Chaining with a Separate Overflow Area

Move overflow records to a **Separate Overflow Area**.

A linked list of synonyms start at their home address in the Primary data area, continuing in the separate overflow area.

Example X, with separate overflow area:

primary data area

20	ADAMS	0
21	BATES	1
22		
23		
24	EVANS	-1
25		

overflow area

0	COLES	2
1	DEAN	-1
2	FLINT	-1
3		
	⋮	⋮

When the packing density is higher than 1 an overflow area is **required**.

4) Scatter Tables: Indexing Revisited

Similar to chaining with separate overflow, but the hashed file contains no records, but only pointers to data records. The scatter

Example X organized as scatter table:

index (hashed)

	⋮
20	0
21	1
22	
23	
24	4
	⋮

datafile (entry-sequenced, sorted, etc.)

	data	next
0	ADAMS	2
1	BATES	3
2	COLES	5
3	DEAN	-1
4	EVANS	-1
5	FLINT	-1

Note that the data file can be organized in many different ways: sorted file, entry sequenced file, etc.

Patterns of Record Access

Twenty percent of the students send 80 percent of the e-mails. *L. M.*

Using knowledge of pattern of record access to improve performance ...

Suppose you know that 80% of the searches occur in 20% of the items.

How to use this info to try to reduce search length in a hashed file ?

- Keep track of record access for a period of time (say 1 month).
- Sort the file in descending order of access.
- Re-hash using this order.

Records more frequently searched are more likely to be **at** or **close to** their home addresses.

LECTURE 18: EXTENDIBLE HASHING I

Contents of today's lecture:

- What is extendible hashing.
- Insertions in extendible hashing.

Reference: FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Sections 12.1,12.2,12.3(overview only).

What is extendible hashing ?

- It is an approach that tries to make hashing **dynamic**, i.e. to allow insertions and deletions to occur without resulting in poor performance after many of these operations.
  Why this is not the case for ordinary hashing?
- Extendible hashing combines two ingredients:
  hashing and **tries**.
  (tries are digital trees like the one used in Lempel-Ziv)
- Keys are placed into buckets, which are independent parts of a file in disk.
  Keys having a hashing address with the same prefix share the same bucket.
  A trie is used for fast access to the buckets. It uses a prefix of the hashing address in order to locate the desired bucket.

Tries and buckets

Consider the following grouping of keys into buckets, depending on the prefix of their hash addresses:

bucket:	this bucket contains keys with hash address with prefix:
A	0
B	10
C	11

Drawing of the trie that provides and index to buckets:

```
                         +-------+
          *------------> |   A   |
    0                    +-------+


   *
                         +-------+
    1       0  *-------> |   B   |
                         +-------+
         *
                         +-------+
            1  *-------> |   C   |
                         +-------+
```

Note: You need to connect the parents to the children in the drawing of the trie above.

Directory structure and buckets

Representing the trie as a tree would take too much space. Instead, do the following:

1) Extend the tree to a complete binary tree:

```
            0  *----+
                    +--> +-------+
        *                |   A   |
   0                +--> +-------+
            1  *----+
  *
                         +-------+
   1        0  *-------> |   B   |
                         +-------+
        *
                         +-------+
            1  *-------> |   C   |
                         +-------+
```

2) Flatten the trie up into an array:

```
    +-----+       +-------+
 00 |     |------>|   A   |
    +-----+   +-->+-------+
 01 |     |---+
    +-----+       +-------+
 10 |     |------>|   B   |
    +-----+       +-------+
 11 |     |--+
    +-----+  |    +-------+
             +--->|   C   |
                  +-------+
```

How to search in extendible hashing ?

Searching for a key:

- Calculate the hash address of the key (note that no table size is specified, so we don't take "mod").
- Check how many bits are used in the directory (2 bits in the previous example). Call i this number of bits.
- Take the least significative i bits of the hash address (in reverse order). This gives an index of the directory.
- Using this index, go to the directory and find the bucket address where the record might be.

What makes it extendible?

So far we have not discussed how this approach can be dynamic, making the table expand or shrink as records are added or deleted. The dynamic aspects are handled by two mechanisms:

- Insertions and bucket splitting.
- Deletions and bucket combination.

Bucket splitting to handle overflow

Extendible hashing solves bucket overflow by splitting the bucket into two and if necessary increasing the directory size.
When the directory size increases it doubles its size a certain number of times.

Example:

To simplify matters, let us assume the keys are numbers and the hash function returns the number itself.
For instance, $h(20) = 20$.
Bucket size: 2

numbers	1	2	3	5	6	7
binary representation:	0001	0010	0011	0101	0110	0111

```
   +------+       +-------+
00 |      |------>|  2, 6 |
   +------+   +-->+-------+
01 |      |---+
   +------+       +-------+
10 |      |------>|  1, 5 |
   +------+       +-------+
11 |      |--+
   +------+  |    +-------+
             +--->|  3, 7 |
                  +-------+
```

```
   +-----+       +-------+
00 |     |------>|  2, 6 |
   +-----+   +-->+-------+
01 |     |---+
   +-----+       +-------+
10 |     |------>|  1, 5 |
   +-----+       +-------+
11 |     |--+
   +-----+  |    +-------+
            +--->|  3, 7 |
                 +-------+
```

Insert key 4 into the above structure:

- Since `4` has binary representation 0100, it must go into the first bucket. Since it is full, it will get split.
- The splitting will separate `4` from `2,6` since the last two bits of `4` in reverse order are `00` and the ones of `2,6` are `01`.
- the directory is prepared to accommodate the splitting, so no doubling is needed.

```
                 +-------+
            +--->|   4   |
   +-----+  |    +-------+
00 |     |--+    +-------+
   +-----+   +-->|  2, 6 |
01 |     |---+   +-------+
   +-----+       +-------+
10 |     |------>|  1, 5 |
   +-----+       +-------+
11 |     |--+
   +-----+  |    +-------+
            +--->|  3, 7 |
                 +-------+
```

Bucket splitting with increase in directory size

```
   +-----+       +-------+
00 |     |------>|  2, 6 |
   +-----+   +-->+-------+
01 |     |---+
   +-----+       +-------+
10 |     |------>|  1, 5 |
   +-----+       +-------+
11 |     |--+
   +-----+  |    +-------+
            +--->|  3, 7 |
                 +-------+
```

Insert key `9` into the above structure:

- Since `9` has binary representation 1001, it must go into the bucket indexed by 10. Since it is full, it will get split.
- The splitting will separate `1,9` from `5` since the last two bits of `1,9` in reverse order are `100` and the ones of `5` are `101`.
- The directory is not prepared to accommodate the splitting, so doubling is needed. The doubling in the directory size will add an extra bit to the directory index.

Result of the addition of key `9` as described above:

```
    +------+
000 |      |---+
    +------+   +-->+-------+
001 |      |------>|  2, 6 |
    +------+   +-->+-------+
010 |      |---|
    +------+   |
011 |      |---+   +-------+
    +------+    +->| 1, 9  |
100 |      |----+  +-------+
    +------+       +-------+
101 |      |------>|   5   |
    +------+       +-------+
110 |      |------>+-------+
    +------+   +-->|  3,7  |
111 |      |---+   +-------+
    +------+
```

Insertion Algorithm

- Calculate the hash function for the key and the key address in the current directory.
- Follow the directory address to find the bucket that should receive the key.
- Insert into the bucket, splitting it if necessary.
- If splitting took place, calculate i: the number of bits necessary to differentiate keys within this bucket. Double the directory as many times as needed to create a directory indexed by i bits.

Note: This algorithm does not work if such an i does not exist. That is, if there are too many keys with the same hash address ("too many" here meaning more than the size of a bucket).

Why?

Which are possible fixes?

Practice exercise

Insert the following keys, into an empty extendible hashing structure:

2,10,7,3,5,16,15,9

Show the structure after each insertion.

All the intermediate steps will be performed in lecture.
Final solution for your checking only:

```
000 and 001 point to bucket with key 16
010 and 011 point to bucket with keys 2,10
100 and 101 point to bucket with key 5,9
110 points to bucket with key 3
111 points to bucket with keys 7,15
```

LECTURE 19: EXTENDIBLE HASHING II

Contents of today's lecture:

- Insertions: a closer look at bucket splitting.
- Deletions in extendible hashing.
- Extendible hashing performance.

Reference: FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Sections 12.4,12.5.

Insertions: bucket splitting re-visited

In some cases, several splits are necessary in order to accommodate a new key to be inserted. Let us consider one such case.
Consider the following example:

```
    +-----+
000 |     |---+
    +-----+   +-->+-------+
001 |     |------>|  2,6  |  bucket A
    +-----+   +-->+-------+
010 |     |---|
    +-----+   |
011 |     |---+   +-------+
    +-----+    +->| 1, 9  |
100 |     |----+  +-------+
    +-----+       +-------+
101 |     |------>|   5   |
    +-----+       +-------+
110 |     |------>+-------+
    +-----+   +-->|  3,7  |
111 |     |---+   +-------+
    +-----+
```

Insert key 10
Bucket A must split.
Note that A has depth 1, since one digit (namely 0) determines whether a key should be placed in A.

The first split will create a new bucket A1, so that A and A1 have now depth 2.
With depth 2, two digits will be examined to determine whether a key goes to A (00*) or A1 (01*).
After redistribution, the structure becomes:

```
    +-----+       +-------+
000 |     |------>|(empty)|  bucket A
    +-----+  +--->+-------+
001 |     |--+
    +-----+       +-------+
010 |     |------>|  2,6  | bucket A1
    +-----+   +-->+-------+
011 |     |---+   +-------+
    +-----+    +->| 1, 9  |
100 |     |----+  +-------+
    +-----+       +-------+
101 |     |------>|   5   |
    +-----+       +-------+
110 |     |------>+-------+
    +-----+   +-->|  3,7  |
111 |     |---+   +-------+
    +-----+
```

A becomes empty and A1 get both keys.
After this split, the bucket pointed by 010, namely A1, is still full. So, we must split A1.
Bucket A1 has depth 2. It will split into buckets A1 and A2, both with depth 3.

After redistribution, the structure becomes:

```
    +-----+      +-------+
000 |     |------>|(empty)|
    +-----+  +--->+-------+
001 |     |--+    +-------+
    +-----+   +-->|   2   | bucket A1
    |     |---+   +-------+
010 |     |       +-------+
    +-----+   +-->|   6   | bucket A2
011 |     |---+   +-------+
    +-----+       +-------+
100 |     |------>| 1, 9  |
    +-----+       +-------+
101 |     |---+   +-------+
    +-----+   +-->|   5   |
110 |     |---+   +-------+
    +-----+   +-->+-------+
111 |     |------>|  3,7  |
    +-----+       +-------+
```

Now the bucket pointed by 010 is not full, so that we can insert key 10 there.

After inserting key 10, the structure becomes:

```
    +-----+       +-------+
000 |     |------>|(empty)|
    +-----+  +--->+-------+
001 |     |--+    +-------+
    +-----+   +-->|  2,10 | bucket A1
    |     |---+   +-------+
010 |     |       +-------+
    +-----+   +-->|   6   | bucket A2
011 |     |---+   +-------+
    +-----+       +-------+
100 |     |------>| 1, 9  |
    +-----+       +-------+
101 |     |---+   +-------+
    +-----+   +-->|   5   |
110 |     |---+   +-------+
    +-----+   +-->+-------+
111 |     |------>|  3,7  |
    +-----+       +-------+
```

Insertion Algorithm

```
Insert (key) {

 indexKey = index in the directory where key
            should be placed
 Let A = bucket pointed by indexKey

 if (bucket A is not full) then
     Insert key into bucket A
 else  { // bucket is full so must split
   if (bucket depth is equal to directory depth) then
       { double directory size,
         re-adjusting bucket pointers;
       }
   split bucket A;
   Insert (key);  // recursive call to itself
   }
}
```

Deletions in Extendible Hashing

When we delete a key we may be able to **combine buckets**.

After the bucket combination, the directory may or may not be **collapsed**.

Combination and collapsing is a recursive process.

Bucket combination

A bucket may have a **buddy bucket**, that is a bucket that may be combined to it.
In the following example, which of the following buckets is such that when deleting a key from it, the bucket can be combined to another bucket?

```
                          bucket size = 2 records
   +-----+
000 |     | -> bucket 1 (1 record)
   +-----+
001 |     | -> bucket 2 (2 records)
   +-----+
010 |     |--+
   +-----+  +--> bucket 3 (1 record)
011 |     |--+
   +-----+
100 |     | -> bucket 4  (1 record)
   +-----+
101 |     | -> bucket 5  (2 record)
   +-----+
110 |     | -> bucket 6  (2 record)
   +-----+
111 |     | -> bucket 7  (2 record)
   +-----+
```

Answer: buckets 1&2, buckets 4&5
Why not buckets 3, 6, 7 ?

Buddy buckets

One bucket can only be combined with its **buddy bucket**.

A bucket and its buddy bucket must be distinct buckets pointed to by sibling nodes in the directory trie.

The index of the buddy bucket is obtained by switching the last bit of the bucket's index.

How to test if the bucket pointed out by index i has a buddy bucket?

```
i = 101
swap the last bit of i: iSwap = 100
if  (directory[i] != directory[iSwap]) then
     bucket pointed by direct[iSwap]
     is the buddy bucket
else there is no buddy bucket.
```

Directory collapsing

We can collapse a directory if every pair of "sibling" indexes point to the same bucket.

Check if we can collapse a directory as follows:

```
canColapse = false;
if (directory size is larger than 1) then {
    canColapse = true;
    for i=0 to (size/2 -1) do
      if (directory[2*i] != directory[2*i+1]
      then { canColapse = false;
             exit for;
           }
}
```

If `canColapse` is true then we can collapse the directory in the following way:

- Create a new directory with half the size
- Copy the pointers from positions 0 to 0, 2 to 1, . . . , (size-2) to (size/2 -1).

Deletion Algorithm

`Delete (key):`

1. Search for `key`.
2. If `key` not found, stop.
3. If `key` was found then remove `key` from its bucket `b` , in the following way:
   - physically remove `key` from bucket `b`
   - run `tryCombine(b)` in order to try to combine buckets and shrink directory.

Description of `tryCombine(b)`:

```
if (b has a buddy bucket c)  then
   if ( b.numkeys + c.numkeys) <= maxkeys {
       combine buckets b and c into bucket b.
       try to collapse the directory;
       if (directory was collapsed) then
          tryCombine(b);
       }
```

Deletion Example

Delete 5 from the following structure:

```
    +------+
000 |      |---+
    +------+   +-->+--------+
001 |      |------>|  2, 6  |
    +------+   +-->+--------+
010 |      |---|
    +------+   |
011 |      |---+   +--------+
    +------+    +->| 1, 9   |
100 |      |----+  +--------+
    +------+       +--------+
101 |      |------>|   5    |
    +------+       +--------+
110 |      |------>+--------+
    +------+   +-->|(empty)|
111 |      |---+   +--------+
    +------+
```

Search for 5 and physically remove it from its bucket `b`:

```
    +-----+
000 |     |---+
    +-----+   +-->+-------+
001 |     |------>|  2, 6 |
    +-----+   +-->+-------+
010 |     |---|
    +-----+   |
011 |     |---+   +-------+
    +-----+    +->| 1, 9  | <<<<< buddy bucket c
100 |     |----+  +-------+
    +-----+       +-------+
101 |     |------>|(empty)|  <<<< bucket b
    +-----+       +-------+
110 |     |------>+-------+
    +-----+   +-->|(empty)|
111 |     |---+   +-------+
    +-----+
```

Run `tryCombine(b)`.

Since `b` contains 0 keys and buddy bucket `c` contains 2 keys, buckets `b` and `c` can be combined...

After combining `c` and `b` we get:

```
    +-----+
000 |     |---+
    +-----+   +-->+-------+
001 |     |------>|  2, 6 |
    +-----+   +-->+-------+
010 |     |---|
    +-----+   |
011 |     |---+
    +-----+
100 |     |------>+-------+
    +-----+       |  1, 9 |  b = buckets b,c combined
101 |     |------>+-------+
    +-----+
110 |     |------>+-------+
    +-----+   +-->|(empty)|
111 |     |---+   +-------+
    +-----+
```

The directory can be collapsed...

The collapsed directory:

```
   +-----+       +-------+
00 |     |---+-->|  2, 6 |
   +-----+   |   +-------+
01 |     |---+
   +-----+       +-------+
10 |     |------>|  1, 9 | bucket b
   +-----+       +-------+
11 |     |------>+-------+
   +-----+       |(empty)| buddy bucket c
                 +-------+
```

The recursive call to `tryCombine(b)` now combines `b` and its buddy bucket `c`:

```
   +-----+       +-------+
00 |     |---+-->|  2, 6 |
   +-----+   |   +-------+
01 |     |---+
   +-----+       +-------+
10 |     |---+-->|  1, 9 | b = buckets b, c combined
   +-----+   |   +-------+
11 |     |---+
   +-----+
```

The directory can be collapsed again...

The collapsed directory:

```
    +-----+      +-------+
  0 |     |---+-->|  2, 6 | buddy bucket c
    +-----+      +-------+
  1 |     |--+   +-------+
    +-----+  +--->|  1, 9 | bucket b
                 +-------+
```

The next recursive call to `tryCombine(b)` would detect that `b` cannot be combined with its buddy bucket `c`.

The deletion operation has been completed.

Extendible Hashing Performance

Time Performance (Worst Case):

operation	directory kept in main memory	directory kept in disk
search	1 disk access	2 disk accesses
insertion $d=$ dir size after insertion	$O(\log d)$ disk accesses	$O(d)$ disk accesses
deletion $d=$ dir size before deletion	$O(\log d)$ disk accesses	$O(d)$ disk accesses

The great advantage of extended hashing is that its search time is truly $O(1)$, independently from the file size.

In ordinary hashing, this complexity depends on the packing density, which could change after many insertions.

Space Performance:

1. **Space Utilization for Buckets**

   Here we describe results found by Fagin, Nievergelt, Pippinger and Strong (1979).

   Experiments and algorithm analysis by these authors have shown that the packing density for extendible hashing (space utilization for buckets) fluctuates between 0.53 and 0.93.

   They have also shown that, if:

   r = number of records
   b = block size
   N = average number of buckets

   Then,

   $$N \approx \frac{r}{b \cdot ln2}.$$

   Therefore,

   packing density = $\frac{r}{b \cdot N} \approx ln2 \approx 0.69$.

   So, we expect the average bucket utilization to be 69%.

2. **Space Utilization for the Directory**

Given r keys spread over some buckets, what is the expected size of the directory, assuming random keys are inserted into the table?

Flajolet (1983) addressed this problem by doing a careful analysis, in order to estimate the directory size.

The following table shows his findings:

b	5	10	20	...	200
r					
10^3	1.50 K	0.30 K	0.10 K		0.00 K
10^4	25.60 K	4.80 K	1.70 K		0.00 K
10^5	424.10 K	68.20 K	16.80 K		1.00 K
10^6	6.90 M	1.02 M	0.26 M		8.10 K
10^7	112.11 M	12.64 M	2.25 M		0.13 M

1 K = 10^3, 1 M = 10^6. (From Flajolet 1983)

LECTURE 20: B-TREES I

Contents of today's lecture:

- Introduction to multilevel indexing and B-trees.
- Insertions in B trees.

Reference: FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Sections 9.1-9.6.

Introduction to Multilevel Indexing and B-Trees

Problems with **simple indexes** that are kept in disk:

1. Seeking the index is still slow (binary searching):

   We don't want more than 3 or 4 seeks for a search.
   So, here $\log_2(N+1)$ is still slow:

N	$\mathbf{log_2(N+1)}$
15 keys	4
1,000	$\sim$10
100,000	$\sim$17
1,000,000	$\sim$20

2. Insertions and deletions should be as fast as searches:
   In simple indexes, insertion or deletion take O(n) disk accesses (since index should be kept sorted)

Indexing with Binary Search Trees

We could use **balanced** binary search trees:

- **AVL Trees**

  Worst-case search is 1.44 $\log_2$(N+2)

  1,000,000 keys $\rightarrow$ 29 levels

  Still prohibitive...

- **Paged Binary Trees**

  Place subtrees of size K in a single page

  Worst-case search is $\log_{K+1}$(N+1)

K=511, N=134,217,727

Binary trees: 27 seeks
Paged binary tree: 3 seeks

This is good but there are lots of difficulties in **maintaining** (doing insertions and deletions in) a paged binary tree.

Multilevel Indexing

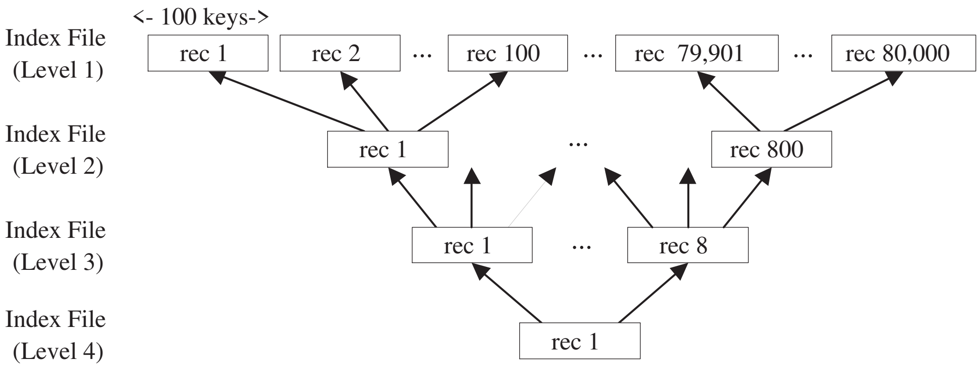
Consider our 8,000,000 example with keysize = 10 bytes.
Index file size = 80 MB
Each **record** in the **index** will contain 100 pairs (key, reference)
A **simple index** would contain: 80,000 records. Too expensive to search (still $\sim$ 16 seeks)

Multilevel Index

Build an index of an index file:

How:

- Build a simple index for the file, sorting keys using the method for external sorting previously studied.
- Build an index for this index.
- Build another index for the previous index, and so on.



Note: That the index of an index stores the largest key in the record it is pointing to.

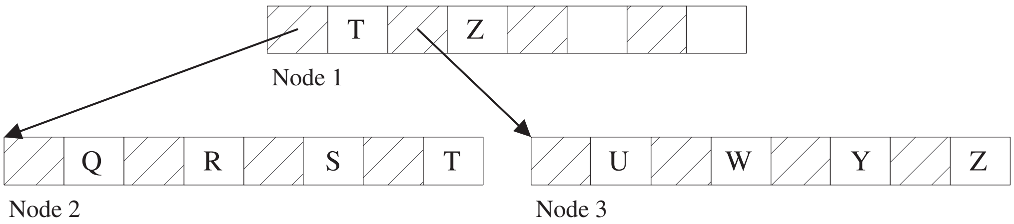
B-Trees - Working Bottom-Up

- Again an index record may contain 100 keys.
- An index record may be half full (each index record may have from 50 to 100 keys).
- When insertion in an index record causes it to overfull:

  - Split record in two

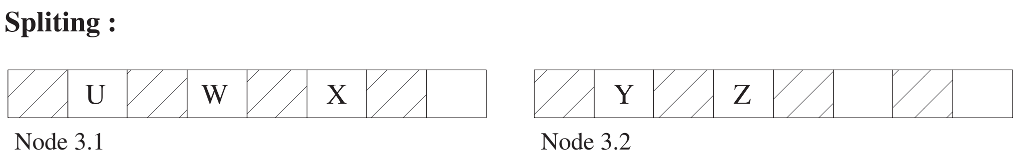
  - "Promote" the largest key in one of the records to the upper level

Example for order = 4 (instead of 100).

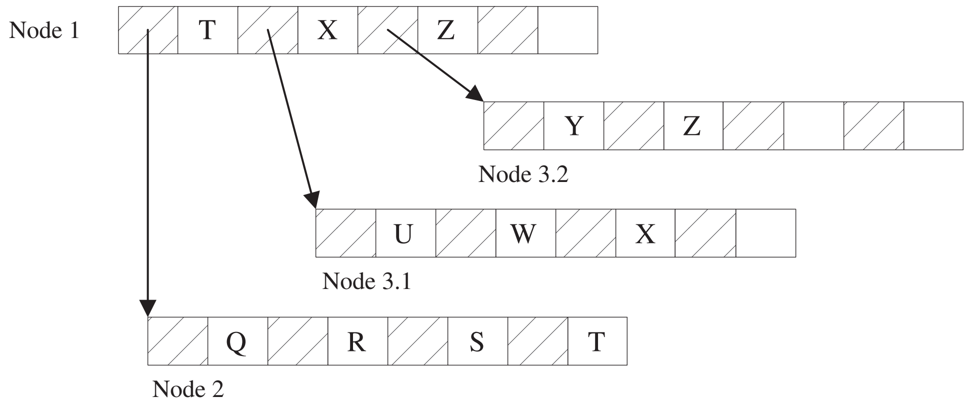


Inserting X

X is between T and Z: insertion in node 3 splits it and generates a promotion of node X.



Promoting largest of Node 3.1.

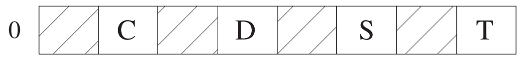


Important: If Node 1 was full, this would generate a new split-promotion of Node 1. This could be propagated up to the root.

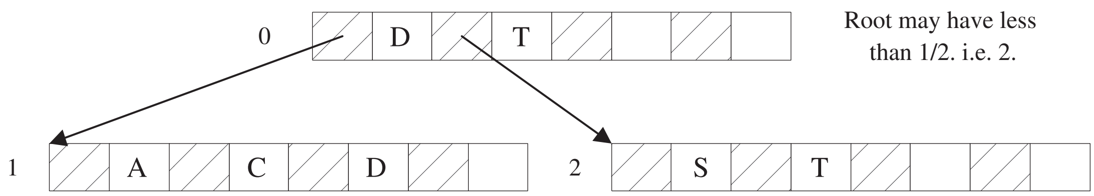
An example showing insertions:

Inserting keys: order = 4

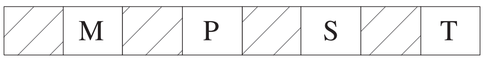
C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J, Y, Q, Z, F, X, V



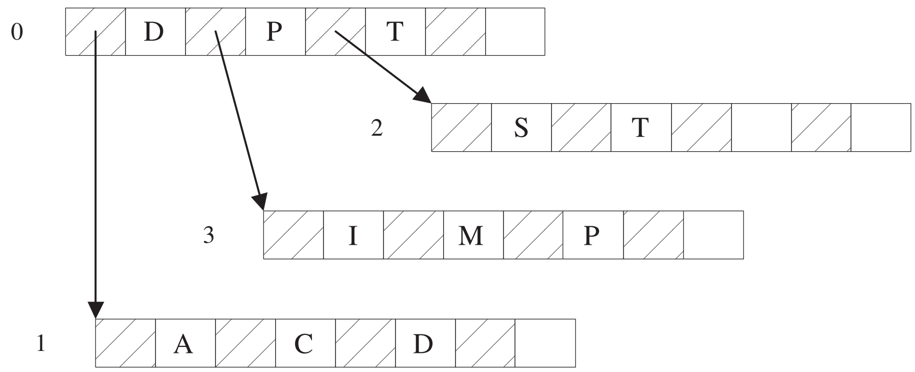
Inserting A: Split and promotion



Inserting M,P only changes Node 2 to :



But inserting “I” Splits and promotes “P”.



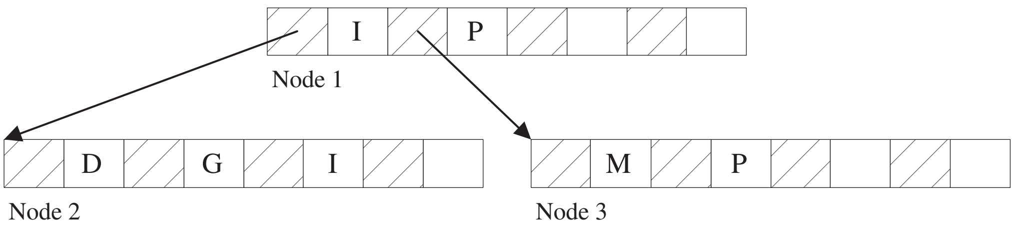
There is room for one more node at level 2. After that the root may get split!

Complete the above example as an exercise.

Important: At each level, no more than 2 nodes are affected. We got search and updates with cost equal to the height of the tree !!!

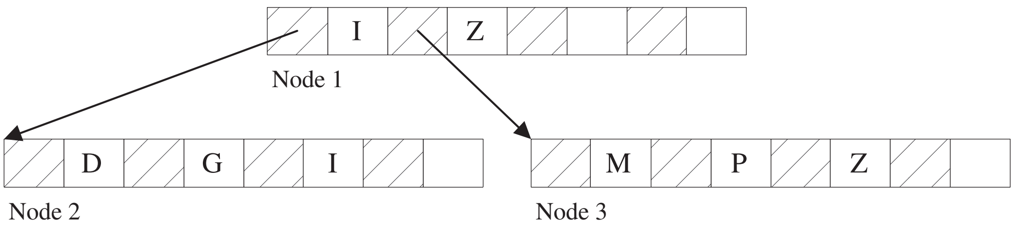
Note regarding insertions in B-trees

Special case of larger key:



Inserting Z

Z is larger than P but P is larger in Node 1, so the place for Z is Node 3.



B-Tree Properties

Properties of a B-tree of order m:

1. Every node has a maximum of m children.
2. Every node, except for the root and the leaves, has at least $\lceil m/2 \rceil$ children.
3. The root has at least two children (unless it is a leaf).
4. All the leafs appear on the same level.
5. The leaf level forms a complete index of the associated data file.

Worst-case search depth

The worst-case depth occurs when every node has the minimum number of children.

Level	Minimum number of keys (children)
1 (root)	2
2	$2 \cdot \lceil m/2 \rceil$
3	$2 \cdot \lceil m/2 \rceil \cdot \lceil m/2 \rceil = 2 \cdot \lceil m/2 \rceil^2$
4	$2 \cdot \lceil m/2 \rceil^3$
...	...
d	$2 \cdot \lceil m/2 \rceil^{d-1}$

If we have N keys in the leaves:

$$N \geq 2 \cdot \lceil m/2 \rceil^{d-1}$$

So, $d \leq 1 + \log_{m/2}(N/2)$

For $N = 1,000,000$ and order $m = 512$, we have

$$d \leq 1 + \log_{256} 500,000$$
$$d \leq 3.37$$

There is at most 3 levels in a B-tree of order 512 holding $1,000,000$ keys.

LECTURE 21: B-TREES II

Contents of today's lecture:

- Outline of `Search` and `Insert` algorihtms
- Deletions in B trees.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 9.8, 9.9, 9.10, 9.11, 9.12

Outline of Search and Insert algorithms

`Search (keytype key)`

1) Find leaf: find the leaf that could contain `key`, loading all the nodes in the path from root to leaf into an array in main memory.
2) Search for `key` in the leaf which was loaded in main memory.

`Insert (keytype key, int datarec_address)`

1) Find leaf (as above).

2) Handle special case of new largest key in tree: update largest key in all nodes that have been loaded into main memory and save them to disk.

3) Insertion, overflow detection and splitting on the update path:
`currentnode` = leaf found in step 1)
`recaddress` = `datarec_address`

   3.1) Insert the pair (`keys, recaddress`) into `currentnode`.

   3.2) If it caused overflow

   - Create `newnode`
   - Split contents between `newnode, currentnode`
   - Store `newnode, currentnode` in disk
   - If no parent node (root), go to step 4)
   - `currentnode` becomes parent node
   - `recaddress` = address in disk of `newnode`
   - `key` = largest key in new node
   - Go back to 3.1)

4) Creation of a new root if the current root was split.

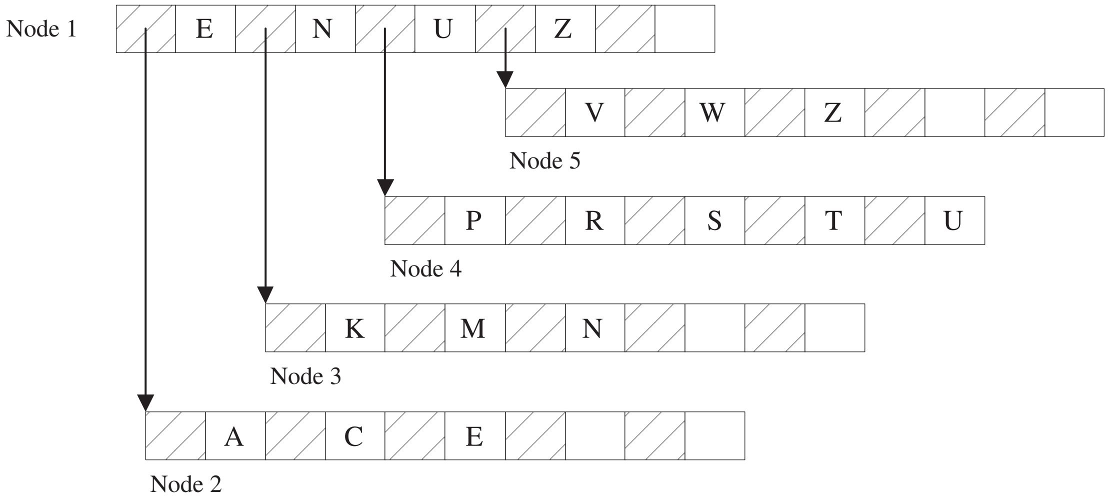
   Create root node pointing to `newnode` and `currentnode`. Save new root to disk.

Deletions in B-Trees

The rules for deleting a key K from a node n in a B-tree:

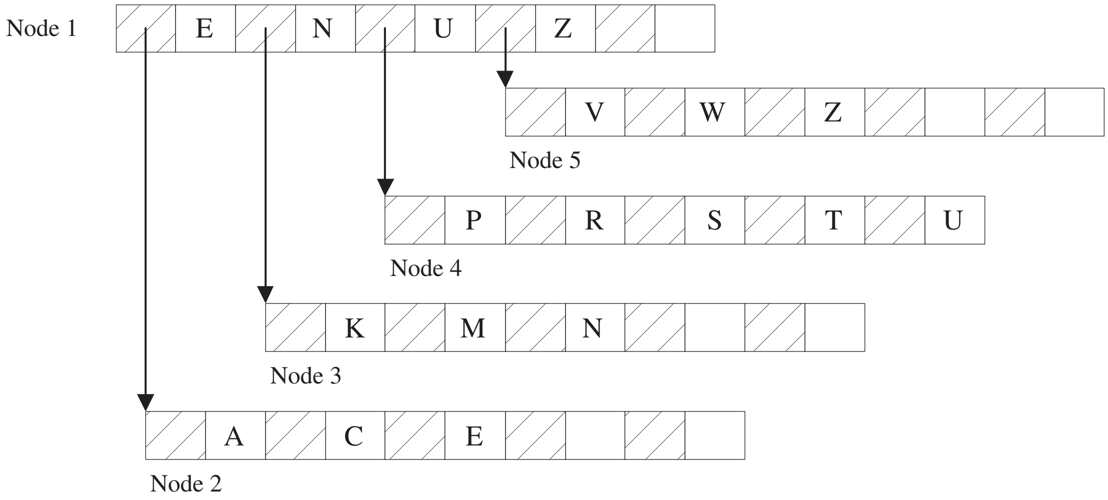
1. If n has more than the minimum number of keys and K is not the largest key in n, simply delete K from n.
2. If n has more than the minimum number of keys and K is the largest key in n, delete K from n and modify the higher level indexes to reflect the new largest key in n.
3. If n has exactly the minimum number of keys and one of the siblings has "few enough keys", **merge** n with its sibling and delete a key from the parent node.
4. If n has exactly the minimum number of keys and one of the siblings has extra keys, **redistribute** by moving some keys from a sibling to n, and modify higher levels to reflect the new largest keys in the affected nodes.

Consider the following example of a B-tree of **order 5** (minimum allowed in node is 3 keys)

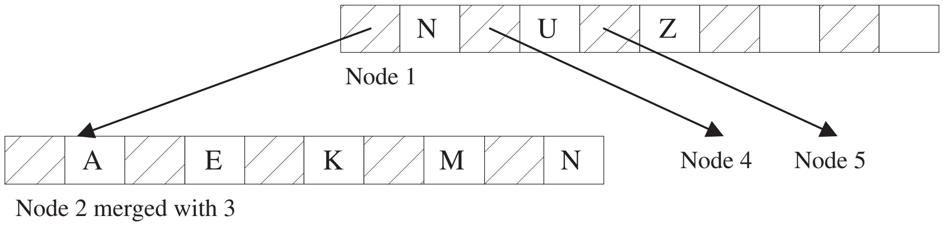


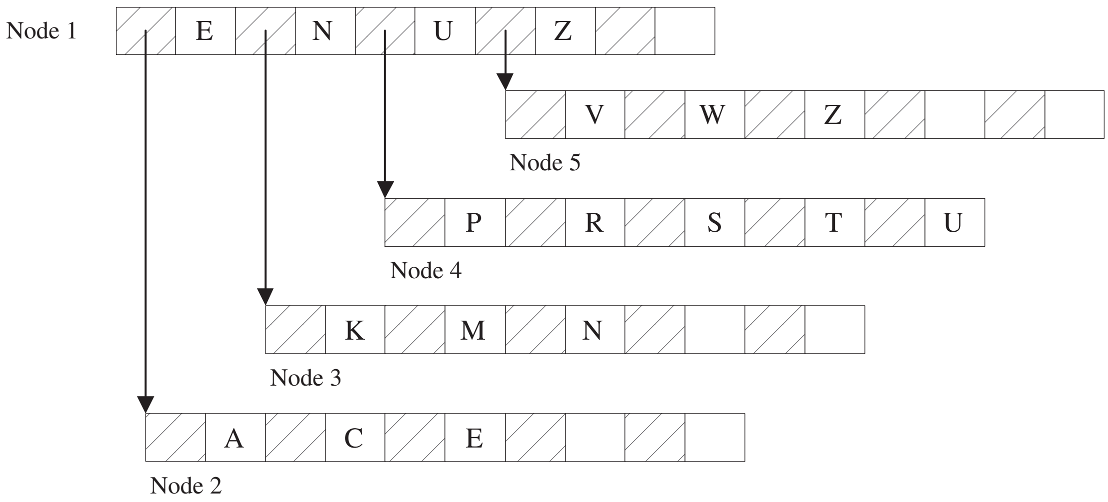
We consider the following 5 alternative modifications on the previous tree:

- Deleting "T" falls into case 1)
- Deleting "U" falls into case 2)

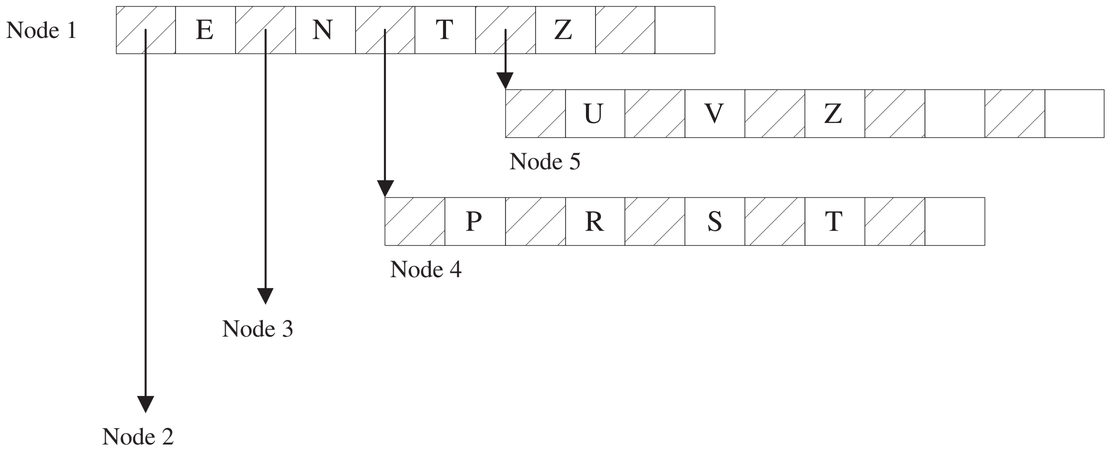


- Deleting "C" falls into case 3) merging with sibling:





- Deleting “W” falls into case 4) redistribution with sibling:



- Deleting “M” allows for two possibilities: case 3) or 4)
  - Merge Node 3 with Node 2; or
  - Redistribute keys between Node 3 and Node 4

Note that “sibling” here refers only to nodes that have the same parent and are **next to each other**.

LECTURE 22: B+ TREES I

Contents of today's lecture:

- Maintaining a sequence set.
- A simple prefix B+ tree.

Reference : Folk, Zoellick and Riccardi, File Structures, 1998. Sections 10.1 - 10.5

Motivation

Some applications require two views of a file :

Indexed view :	**Sequential view :**
Records are indexed by a key	Records can be sequentially accessed in order by key
Direct, indexed access	Sequential access (physically contiguous records)
Interactive, random access	Batch processing (Ex: co-sequential processing)

Example of applications

- Student record system in a university :
  - Indexed view : access to individual records
  - Sequential view : batch processing when posting grades or when fees are paid
- Credit card system :
  - Indexed view : interactive check of accounts
  - Sequential view : batch processing of payment slips

We will look at the following two aspects of the problem :

1. Maintaining a **sequence set** : keeping records in sequential order
2. Adding an **index set** to the sequence set

Maintaining a Sequence Set

Sorting and re-organizing after insertions and deletions is out of question.
We organize the sequence set in the following way:

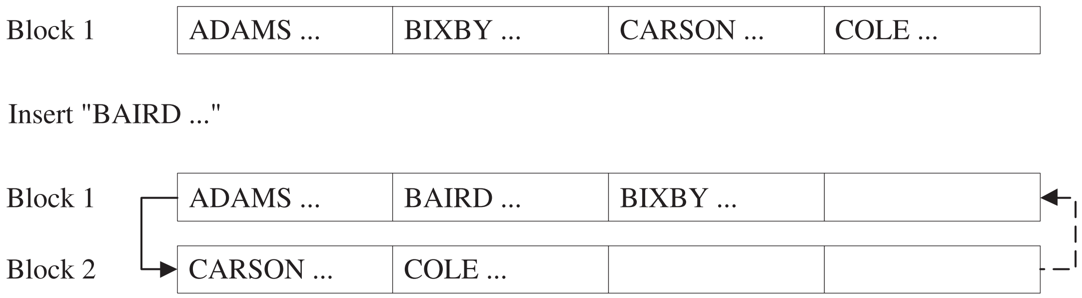
- Records are grouped in **blocks**
- Blocks should be **at least half full**.
- **Link fields** are used to point to the preceding block and the following block (similarly to doubly linked lists)
- Changes (insertion/deletion) are localized into blocks by performing :
  - **Block Splitting** when **insertion** causes overflow
  - **Block Merging** or **Redistribution** when **deletion** causes underflow

Example:

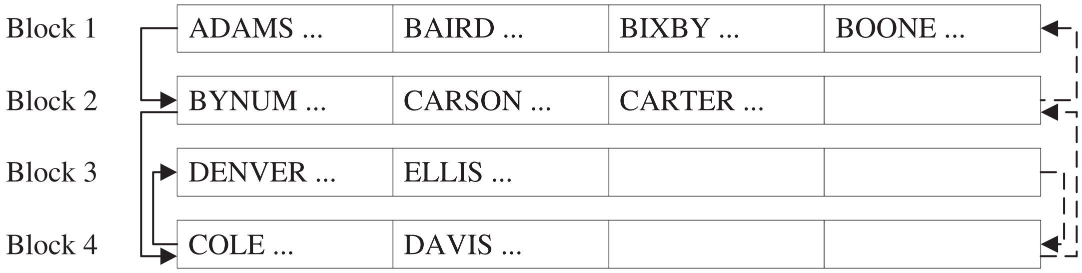
Block size $= 4$

key : **Last Name**

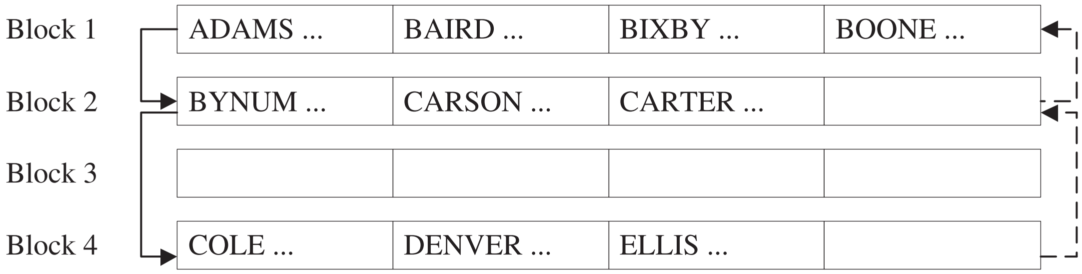
- **Insertion with overflow:**



- **Deletion with merging:**



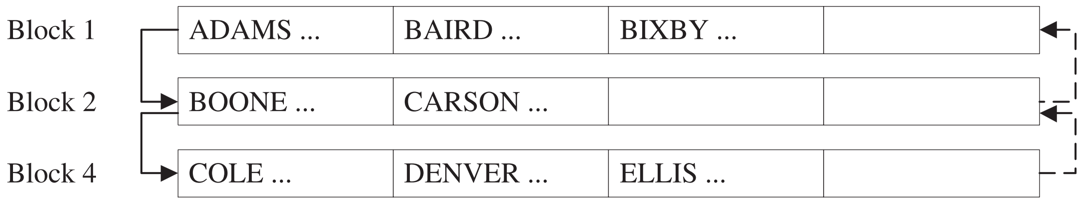
Delete "DAVIS ..." (Merging)



Block 3 is available for re-use

Delete 'BYNUM':
Just remove it from Block 2
Then, delete 'CARTER' :
We can either merge Block 2 and 4 or redistribute records among Blocks 1 and 2.

- **Deletion with redistribution:**



When previous and next blocks are full then redistribution is the only option.

Advantages and disadvantages of the scheme described

Advantages:

- No need to re-organize the whole file after insertions/deletions.

Disadvantages:

- File takes more space than unblocked files (since blocks may be half full).
- The order of the records is not necessarily **physically** sequential (we only guarantee physical sequentiality within a block).

Choosing Block Size

Consider :

- Main memory constraints (must hold at least 2 blocks)
- Avoid seeking within a block (Ex: in sector formatted disks choose block size equal to cluster size).

Adding an Index Set to the Sequential Set

Index will Contain Separators Instead of Keys

Choose the **Shortest Separator** (a prefix)

```
Block     Range of Keys  Separator
-----     -------------  ---------
  1       ADAMS - BERNE
                         BO
  2       BOLEN - CAGE
                         CAM
  3       CAMP  - DUTTON
                         E
  4       EMBRY - EVANS
                         F
  5       FABER - FOLK
                         FOLKS
  6       FOLKS - GADDIS
```

How can we find a separator for `key1`= "CAGE" and `key2` ="CAMP"?

Find the smallest prefix of `key2` that is not a prefix of `key1`. In this example, the separator is "CAM".

The Simple Prefix B+ Tree

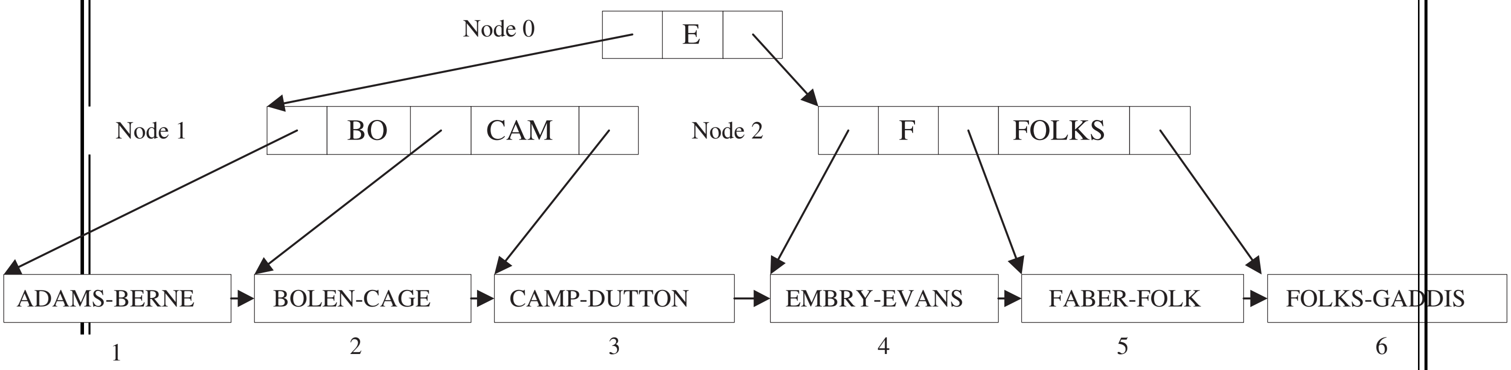
The **simple prefix B+ tree** consists of :

- **sequence set** (as previously seen).
- **index set**: similar to a B-tree index, but storing the shortest separators (prefixes) for the sequence set.

Note : If a node contains N separators, it will contain N+1 children. Using separators slightly modifies the operations in the B-tree index.

Example :

Order of the index set is 3 (i.e. maximum of 2 separators and 3 children). Note: The order is usually much larger, but we made it small for this example.



Search in a simple prefix B+ tree: Search for "EMBRY":

- Retrieve Node 0 (root).
- "EMBRY" > "E", so go right, and retrieve Node 2.
- Since "EMBRY" < "F" go left, and retrieve block number 4.
- Look for the record with key "EMBRY" in block number 4.

LECTURE 23: B+ TREES II

Contents of today's lecture:

- Simple Prefix B+ Tree Maintenance: Insertions and Deletions

Reference : Folk, Zoellick and Riccardi, File Structures, 1998. Sections 10.6 - 10.7 (overview 10.8 - 10.11).

From B trees to B+ trees

A clarification regarding the book's treatment of B-trees and B+ trees: the transition from understanding B trees to understanding the index set of the simple prefix B+ tree may be facilitated by the following interpretation.

- Look at the B + tree index set as a B-tree in which the smallest element in a child is stored at the parent node in order:

key1, pointer1, key2, pointer2, ..., keyN, pointerN.

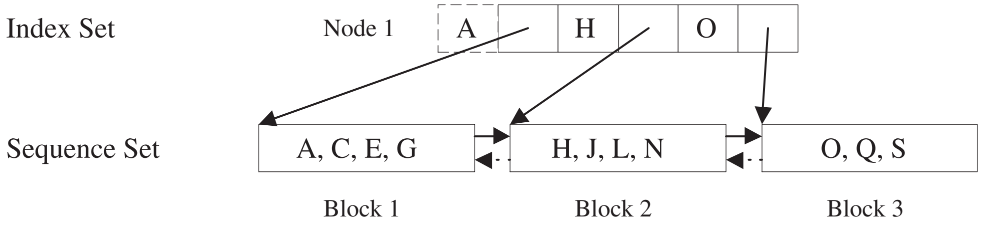
- Then, remember that each key is a separator. And remove key1 from the node's representation. It may help if you think that "key1" is still there, but is "invisible" for understanding purposes.

With this in mind, it should become clear that the updates on the B + tree index set are the same as in regular B-trees.

Simple Prefix B+ Tree Maintenance

Example:

- Sequence set has blocking factor 4
- Index set is a B tree of order 3

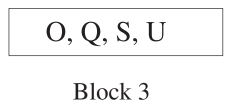


Note that "A" is not really there.

1. Changes which are local to single blocks in the sequence set

   Insert "U":

   - Go to the root
   - Go to the right of "O"
   - Insert "U" to block 3:
     The only modification is



   - There is no change in the index set

Delete “O”:

- Go to the root
- Go to the right of “O”
- Delete “O” from block 3:
  The only modification is

Q, S, U

Block 3

There is no change in the index set: “O” is still a perfect separator for blocks 2 and 3.

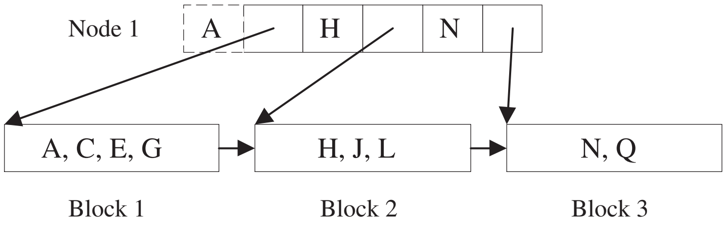
2. Changes involving multiple blocks in the sequence set.

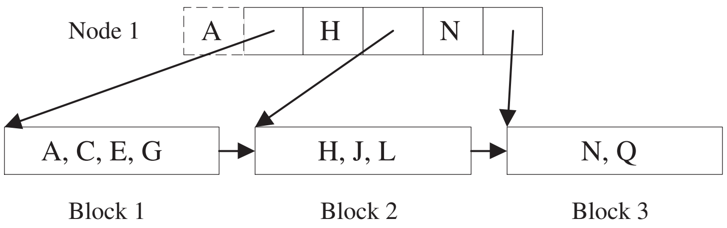
   Delete "S" and "U":

   Now block 3 becomes less than 1/2 full (underflow)

Block 3

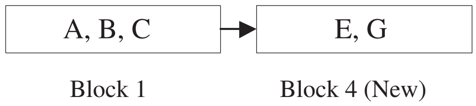
   Since block 2 is full, the only option is **re-distribution** bringing a key from block 2 to block 3:
   We must update the separator "O" to "N".
   The new tree becomes:



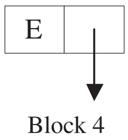


Insert "B":

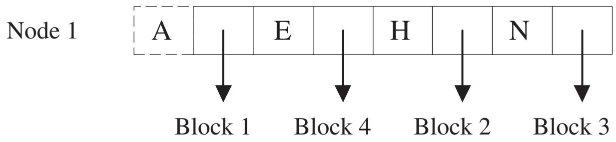
- Go to the root
- Go to the left of "H" to block 1
- Block 1 would have to hold A,B,C,E,G
- Block 1 is split:



Promote new separator "E" together with pointer to new block 4

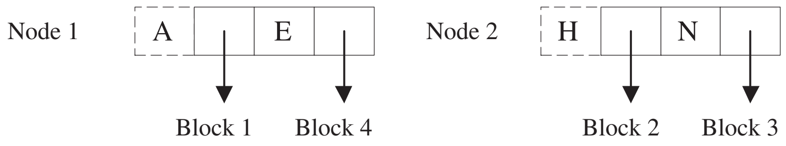


We wished to have:

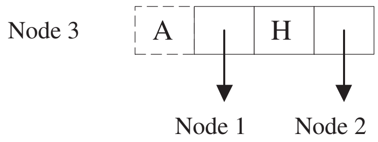


But the order of the index set is 3 (3 pointers, 2 keys).

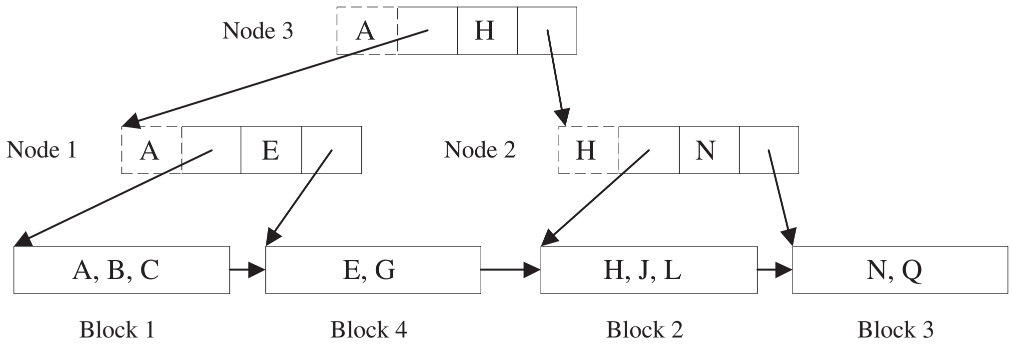
So this causes node to split:



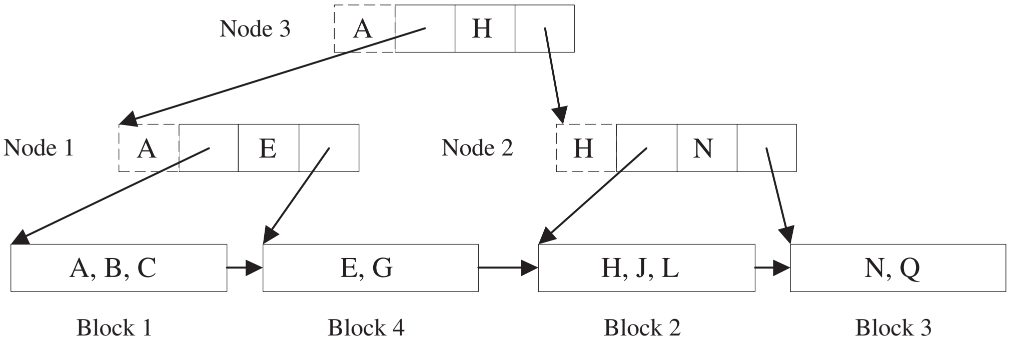
Create a new root to point to both nodes:



The new tree is:

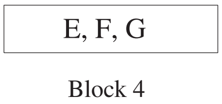


Remember that “A” is not really present in nodes 1 an 3 and that “H” is not really present in node 2.



Insert "F"

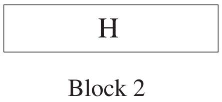
- Go to root
- Go to left of "H"
- Go to right of "E" in Node 1
- Insert "F" in block 4
- Block 4 becomes:



- Index set remains unchanged

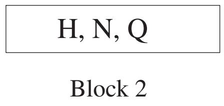
Delete "J" and "L"

Block 2 would become:

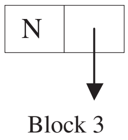


But this is an underflow.
One may get tempted to redistribute among blocks 4 and 2: E, F, G and H would become E, F and G, H.
Why this is not possible ?
Block 4 and block 2 are not siblings! They are cousins.

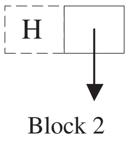
The only sibling of block 2 is block 3.
Redistribution in not possible between H and N,Q, so the only possibility is **merging** blocks 2 and 3 :



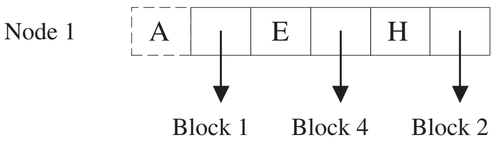
- Send block 3 to `AVAIL LIST`
- Remove the following from node 2



- This causes an underflow in Node 2

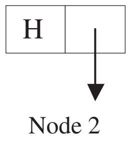


- The only possibility is a merge with its sibling (Node 1):

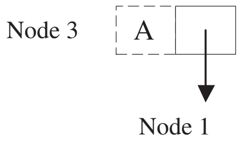


- In our interpretation “H” becomes “visible”; In the textbook’s interpretation “H” is brought down from the root.
- Send node 2 to `AVAIL LIST`

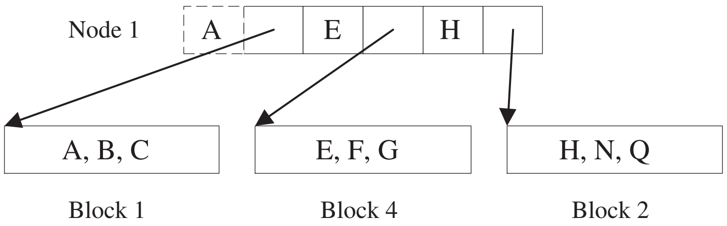
- Now remove



from the root (Node 3). This causes underflow on the root:



- Underflow on the root causes removal of the root (Node 3) and Node 1 becomes the new root :



Blocks were reunited as a big happy family again !!

Compare it with the original tree.

Note : Remember that a B+ tree may be taller, so that splittings or mergings of nodes may propagate for several levels up to the root.

Advanced Observations for Meditation

1. Usually a node for the index set has the same physical size (in bytes) than a block in the sequence set. In this case, we say "index set block" for a node.
2. Usually sequence set blocks and index set blocks are mingled inside the same file.
3. The fact that we use separators of variables length suggests the use of B trees of variable order. The concepts of underflow and overflow become more complex in this case.
4. Index set blocks may have a complex internal structure in order to store variable length separators and allow for binary search on them (see Figure 10.12 on page 442).
5. Building a B+ tree from an existing file already containing many records can be done more efficiently than doing a sequence of insertions into an initially empty B+ tree. This is discussed in Section 10.9 (Building a Simple Prefix B+ Tree).
6. Simple prefix B+ trees or regular B+ trees are very similar. The difference is that the latter stores actual keys rather than shortest separators or prefixes.

LECTURE 24: TO BE INCLUDED

LECTURE 25: COURSE OVERVIEW

Fundamental file processing operations

- open, close, read, write, seek (file as a stream of bytes)

Secondary Storage Devices and System Software

- how different secondary storage devices work (tapes, magnetic disks, CD-ROM)
- the role of different basic software and hardware in I/O (operating system (file manager), I/O processor, disk controller)
- buffering at the level of the system I/O buffers.

Logical view of files and file organization

- file as a collection of records (concepts of records, fields, keys)
- record and field structures
- sequential access; direct access (RRN, byte offset).

Organizing files for performance

- data compression
- reclaiming space in files: handling deletions, AVAIL LIST, etc.
- sorting and searching: internal sorting, binary searching, keysorting.

Cosequential processing

- main characteristics: **sequential access** to input and output files, and **co-ordinated access** of input files.
- main types of processing: matching (intersection) and merging (union).
- main types of application: the merging step in an external sorting method; posting of transactions to a master file.

Indexing

- Primary and secondary key indexes.
- Maintenance of indexed files; different index organizations.
- Inverted lists for secondary indexes.
- Alternative ways of organizing an index:
  - Simple index
    keeping index sorted by key (additions and deletions are expensive: about $O(n)$ disk accesses)
  - B trees and B+ trees
    improvement in time: searches, insertions and deletions in about $O(\log_k n)$ disk accesses, where k is the order of the tree and n is the number of records for B trees or the number of blocks of records for B+ trees.
  - Hashed index: searches, insertions and deletions in constant expected time, i.e. expected time $O(1)$, provided that the hash function disperses the keys well (approximately random).

B trees and B+ trees

- We started from the problem of maintenance of simple indexes.
- B trees: multi-level index that work from bottom up and guarantees searches and updates in about $O(\log_k n)$ disk accesses. More precisely, the worst case search time, insertion time and deletion time (in number of disk accesses) is in the worst-case proportional to the height of the B-tree. The maximum B tree height is $1 + \log_{\lceil k/2 \rceil} n/2$, where n is the number of keys and k is the order of the B tree.
- Then, we discussed the problem of having both an indexed and a sequential view of the file: B+ trees.
- B+ tree = sequence set + index set
  The index set is a B tree; the sequence set is like a doubly linked list of **blocks** of records.
- simple prefix B+ trees: B+ trees in which we store separators (short prefixes) rather than keys, in the index set.
- We learned B trees and B+ trees maintenance and operations.

Hashing and Extedible Hashing

- We have discussed **static hashing** techniques: expected time for access is $O(1)$ when when hash function is good and file doesn't change too much (not many deletions/additions).
- We have discussed: hash functions, record distribution and search length, the use of buckets, collision resolution techniques (progressive overflow, chained progressive overflow, chaining with a separate overflow area, scatter tables, patterns of record access).
- We have discussed **extendible hashing**. In extendible hashing, hashing is modified to become self-adjusting, allowing for the desired expected $O(1)$ access even when the file grows a lot; the address space changes after a lot of insertions or deletions. A good hash function is still required.
- If you cannot predict which one could be a good hash function for a dataset, it is preferable to use a B tree or B+ tree (e.g. general purpose data base management systems). If the hash function is not good, hashing may lead to an $O(n)$ performance (e.g. assignment 3), which is prohibitive, while B trees and B+ trees have a guaranteed worst case performance which is quite good.

What's next: Database Management Systems CSI3317

Different layers:

Database management systems
File systems
Physical storage devices

Each layer hides details of the lower level.
This course focused in **file processing**. In order to do file processing efficiently we studied some key issues concerning **physical storage devices**.
CSI3317 will focus on **database management systems**.

As the number of users and applications in an organization grows, file processing evolves into database processing.
A **database management system** is a large software that maintains the data of an organization and mediates between the data and the application programs.
Each application program asks the DBMS for data, the DBMS figures out the best way to locate the data.
The applications are coded without knowing the physical organization of the data.
File processing is done by the DBMS rather than the application program.
The application program describes the database at a high-level, conceptual view; this high-level description is called a data model.
One of the most popular data models is the relational data model; SQL is a commercially used, standard relational-database language.

Difficulties and issues handled by a DBMS:

- data independence: changes in file organization should not require changes in the application programs; example of common changes: add new fields to records of a file, add an index to a file, add and remove secondary indexes.
- data sharing: care must be taken when several users may be reading and modifying the data.
- data integrity: ensuring consistency of data (when data is updated, related data must be updated)

You you learn a lot about this in CSI 3317 ...