CPSC 535: Advanced Algorithms

Instructor: Dr. Doina Bein

Hash Tables

- A hash table is an ADT that
  - is an *indexed sequence* or *array*, i.e
    - a list of items for which the order does matter (sequence)
    - and supports directly addressing (i.e. accessing any item is done in O(1) steps) (indexed)
  - that supports only the dictionary operations Insert, Search and Delete
- Each element has a key, but instead of using the key as the index of the array directly, the index is computed from the key using a so-called *hash function*
- The domain of values for a hash function is a *hash table*

Hashing

- *Hashing*: If the universe (i.e. the set of possible keys) is large then we store only a relatively small subset of keys, K. Then an element with key k is stored in slot h(k) where *h* is a *hash function*:

$$h: U \rightarrow set\ of\ indices$$

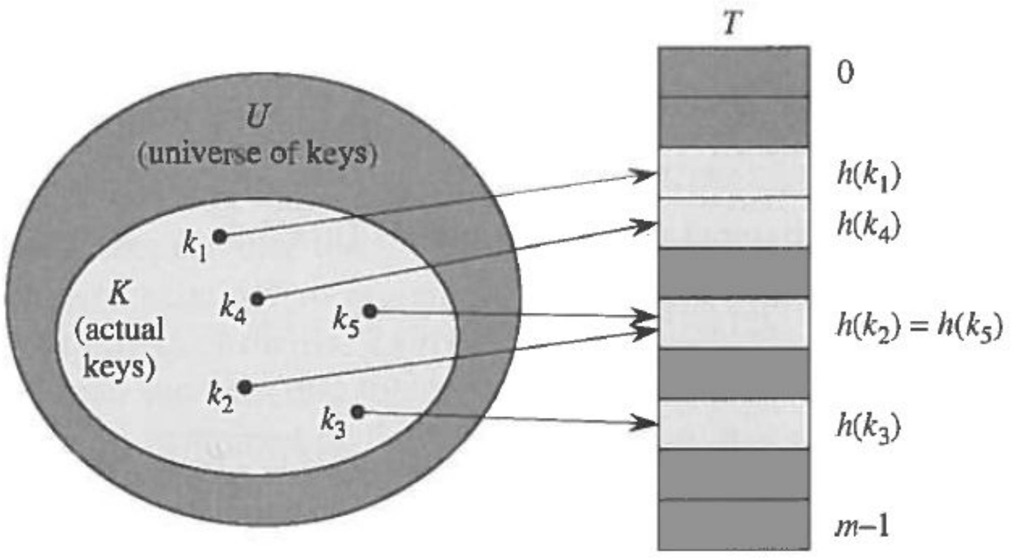


Figure 11.2 Using a hash function h to map keys to hash-table slots. Keys k_2 and k_5 map to the same slot, so they collide.

- element with key k *hashes* to slot h(k)
- h(k) is the *hash value* of key k

Image taken from Cormen's book

Collision

- If two different keys hash to the same hash value then we have a collision
- There are effective ways to resolve collision, the most popular are *chaining* and *open addressing*
- We start with chaining, which is the most frequently used
- Open addressing has gained popularity over the past decade, it will be presented afterwards

Solving Collision by Chaining

- All the elements that share the same hash key are placed in a linked list.
- An entry of the hash table is either NIL or contains the pointer to the first element of the linked list.

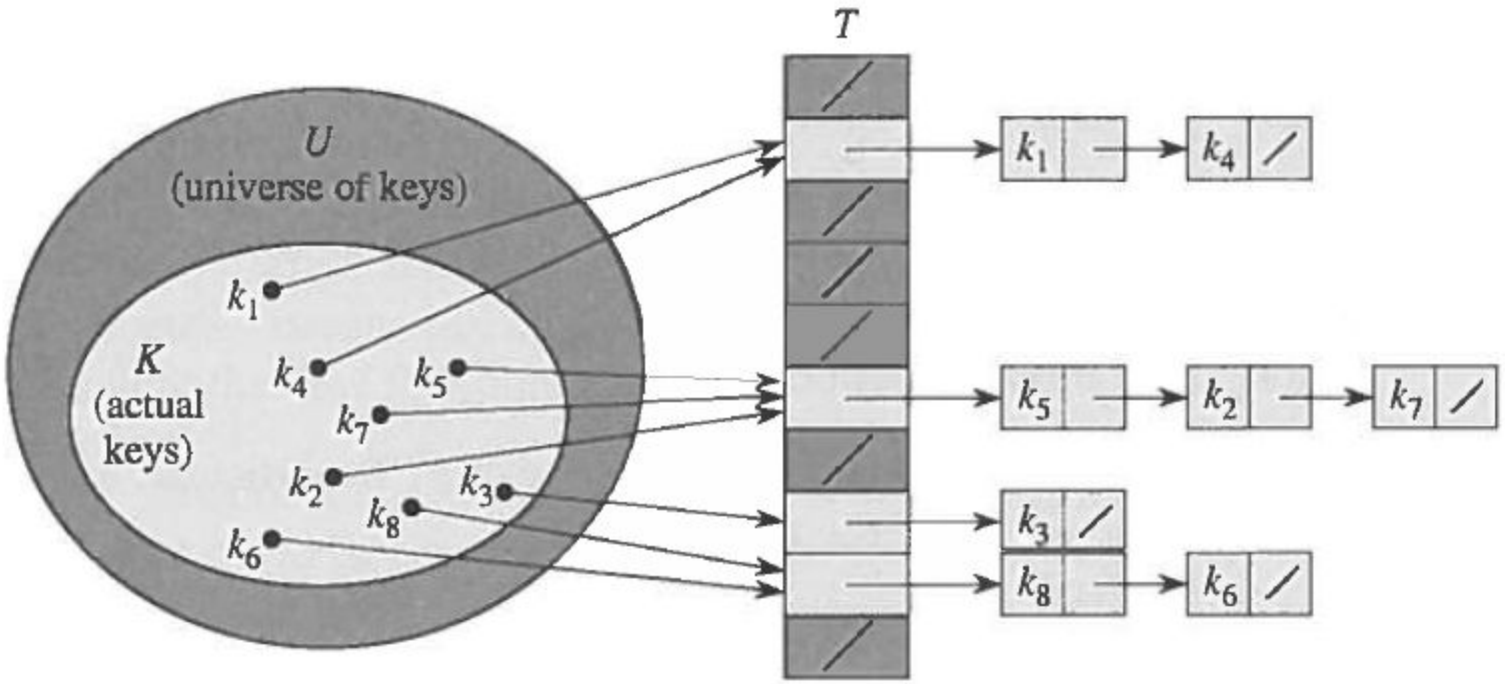


Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

Analysis of Hashing with Chaining

- Let $\mathtt{n}$ be the number of elements stored in the hash table and $\mathtt{m}$ be the number of slots. We define the load factor of the hash table to be $\alpha = {}^{n}/_{m}$
- The dictionary operations (insert, delete, search) are easy to implement
- Insert has O(1): the new element is the first of the list or is inserted at the beginning of the list
- Deletion can take O(1) is the linked list is doubly linked; if the linked list is singly linked, then deletion has the same order as searching
- Searching is proportional to the length of the list
- *Simple uniform hashing*: we assume that the hash value of an element is independent of the hash value of another element
- The expected time of a search (successful or unsuccessful) is $\Theta(1+\alpha)$ under the assumption of simple uniform hashing

Solving Collision by Chaining

- All the elements that share the same hash key are placed in a linked list.
- An entry of the hash table is either NIL or contains the pointer to the first element of the linked list.

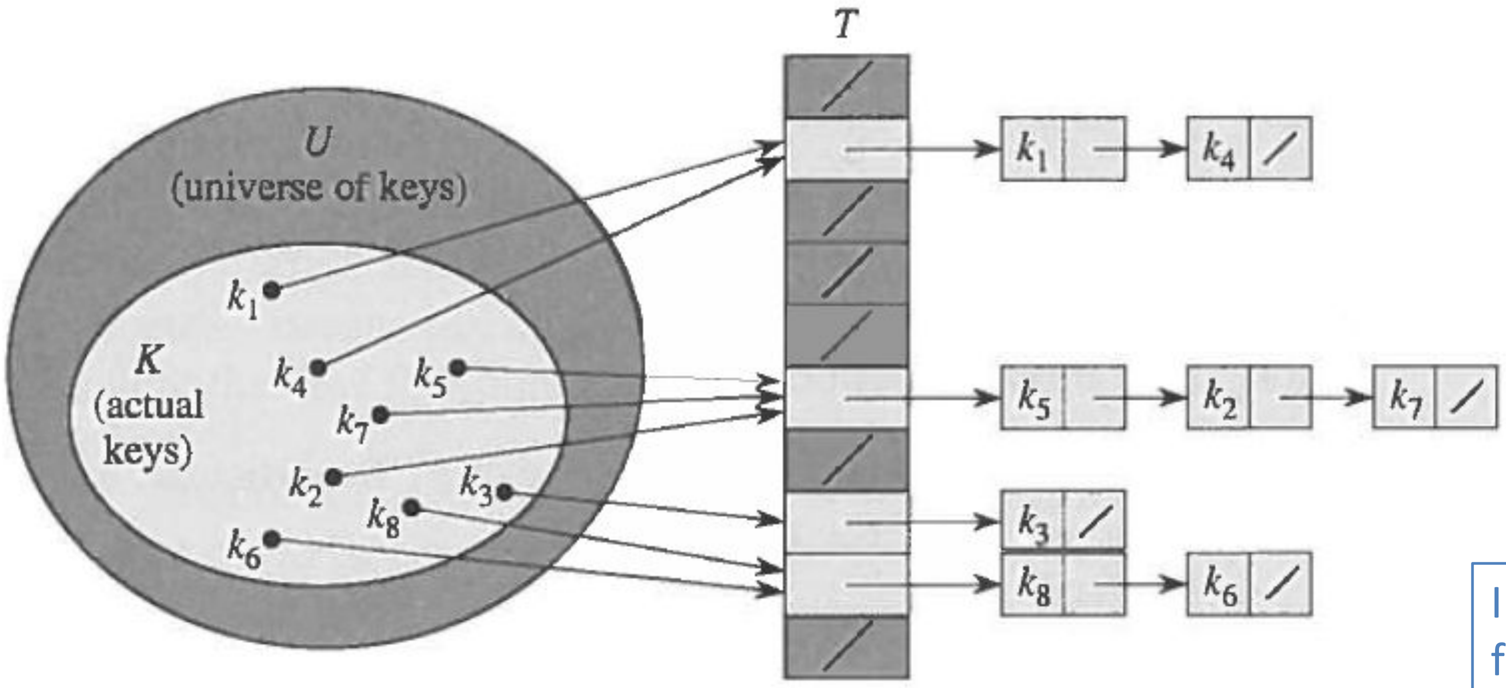


Image taken from Cormen's book

Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

- By not using pointers, we save memory that is used to increase the number of slots in the hash table.
- Insertion: we successively examine, or probe, the hash table until we find an empty slot in which we put the key.
- The sequence of indices to be explored is not $0,1,\ldots,m-1$ which requires `O(m)` search time but depends on the key being inserted.
- We extend the hash function to include the probe number (starting at 0) as a second input value:

$$h\colon U\times\{0,1,2,\cdots,m-1\}\rightarrow\{0,1,2,\cdots,m-1\}$$

- We require that for every key k, the probe sequence $\langle h(k,0),h(k,1),\ldots,h(k,m-1)\rangle$ be a permutation of $\langle 0,1,2,\cdots,m-1\rangle$ so that every hash-table position is eventually considered as a slot for a new key as the table fills up.

Open Addressing

- Linear probing
- Quadratic probing
- Double hashing
- Universal hashing
- Cuckoo hashing

- •A *hash function* maps/compresses messages of arbitrary lengths, typically strings, to a fix length (m-bit output), and is efficiently computable
- –for a given input string, we can get the output in a reasonable amount of time, say, O(n), for an n-bit string
- –output known as the fingerprint or the message digest
- –There’s a generic method called the Merkle-Damgard transform to convert it into a hash function that works on arbitrary-length inputs.
- •What is an example of hash functions?
- –Given a hash function that maps strings to integers in $[0,2^{32}-1]$
- •A hash function is a many-to-one function, not injective, but collisions though unlikely can happen.
- •Hash functions are used to provide time efficient solutions to several problems or applications
- –Good hash functions have few collisions
- –Computationally infeasible to invert
- •*Cryptographic hash functions* are hash functions with additional security requirements
-

Hash Functions

(changed from https://www.cs.purdue.edu/homes/ninghui/courses/555_Spring12/handouts/555_Spring12_topic14.ppt)

- A *hash function* maps/compresses messages of arbitrary lengths, typically strings, to a fix length (m-bit output), and is efficiently computable
  - for a given input string, we can get the output in a reasonable amount of time, say, O(n), for an n-bit string
  - output known as the fingerprint or the message digest
  - There's a generic method called the Merkle-Damgard transform to convert it into a hash function that works on arbitrary-length inputs.
- What is an example of hash functions?
  - Given a hash function that maps strings to integers in $[0,2^{32}-1]$
- A hash function is a many-to-one function, not injective, but collisions though unlikely can happen.
- Hash functions are used to provide time efficient solutions to several problems or applications
  - Good hash functions have few collisions
  - Computationally infeasible to invert
- *Cryptographic hash functions* are hash functions with additional security requirements

Cryptographic Hash Functions

(taken from https://www.lopp.net/pdf/princeton_bitcoin_book.pdf)

- Three additional security properties:

1. Collision resistance: it is infeasible to find two distinct values with the same hash value: $x \neq y$, yet $H(x) = H(y)$
No hash function has been proven collision-free.

2. Hiding: If r is chosen from a probability distribution that has high min-entropy, then given $H(r \,||\, x)$, it is infeasible to find x.
   - *min-entropy* is a measure of how predictable an outcome is
   - ||: concatenation
   - r: a secret random value, called a *nonce*

3. Puzzle-friendly: For every possible output value y, if k is chosen from a distribution with high min-entropy, then it is infeasible to find x such that $H(k \,||\, x) = y$ in a given amount of time.
   - not a general requirement for cryptographic hash functions, but one that will be useful for cryptocurrencies specifically

Collision Resistance

Given a function h: $X \rightarrow Y$, then we say that h is:

- preimage resistant (one-way):

  if given $y \in Y$ it is computationally infeasible to find a value $x \in X$ s.t. $h(x) = y$

- 2-nd preimage resistant (weak collision resistant):

  if given $x \in X$ it is computationally infeasible to find a value $x' \in X$, s.t. $x' \neq x$ and $h(x') = h(x)$

- collision resistant (strong collision resistant):

  if it is computationally infeasible to find two distinct values $x', x \in X$, s.t. $h(x') = h(x)$

Example of collision resistant hash fct

(taken from https://www.lopp.net/pdf/princeton_bitcoin_book.pdf)

- A hash function with a 256-bit output size: pick 2^{256} + 1 distinct values, compute the hashes of each of them, and check if there are any two outputs are equal
- There is at least one collision, but to find it one would have to compute the hash function 2^{256} the worse case and 2^{128} the average case.
- If a computer calculates 10,000 hashes per second, it would take more than one octillion (10^{27}) years to calculate 2^{128} hashes.

Example of hash function with hiding property

- High min-entropy means that the distribution is “very spread out”, so that no particular value is chosen with more than negligible probability.
- In cryptography, the term nonce is used to refer to a value that can only be used once.
- If the nonce r is chosen uniformly from among all of the strings that are 256 bits long, then any particular string was chosen with probability $1/2^{256}$ which is an infinitesimally small value.

Example of hash function that is puzzle friendly

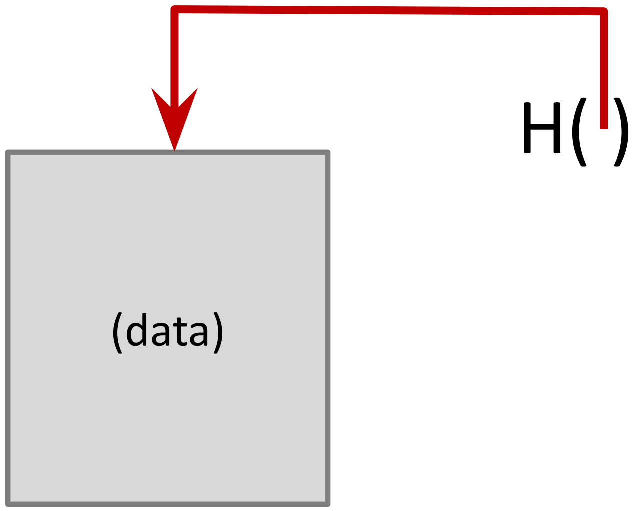
- A *search puzzle* consists of a hash function H, a value r chosen from a high min-entropy distribution and a target set Y, $|Y| << |X|$.
- A solution to this puzzle is a value x such that $H(x||r) \in Y$.
- The size of Y determines how hard the puzzle is.
- If H has an n-bit output, then it can take any of 2^n values. Solving the puzzle requires finding an input so that the output falls within the set Y, which is typically much smaller than the set of all outputs.
- Puzzle-friendly property implies that no solving strategy is much better than trying random values of x.
- If we want to pose a puzzle that's difficult to solve, we can do it this way as long as we can generate r in a suitably random way.
- Bitcoin mining is a sort of computational puzzle.

Using Hashing

- Prevent forgery, modification
  - MAC (Message Authentication Code)
  - Hash as message digest: If we know H(x) = H(y), it’s safe to assume that x = y.
  - To recognize a file that we saw before, just remember its hash and no need to compare word by word.
    - DropBox auto synchronizes modified files.
- Data Structures:
  - Hash list
  - Hash (Merkle) Tree
  - Hash Table
- Hash table vs Hash list vs Hash tree? Article on StackOverflow: https://stackoverflow.com/questions/2974597/hash-table-vs-hash-list-vs-hash-tree

Hash pointer

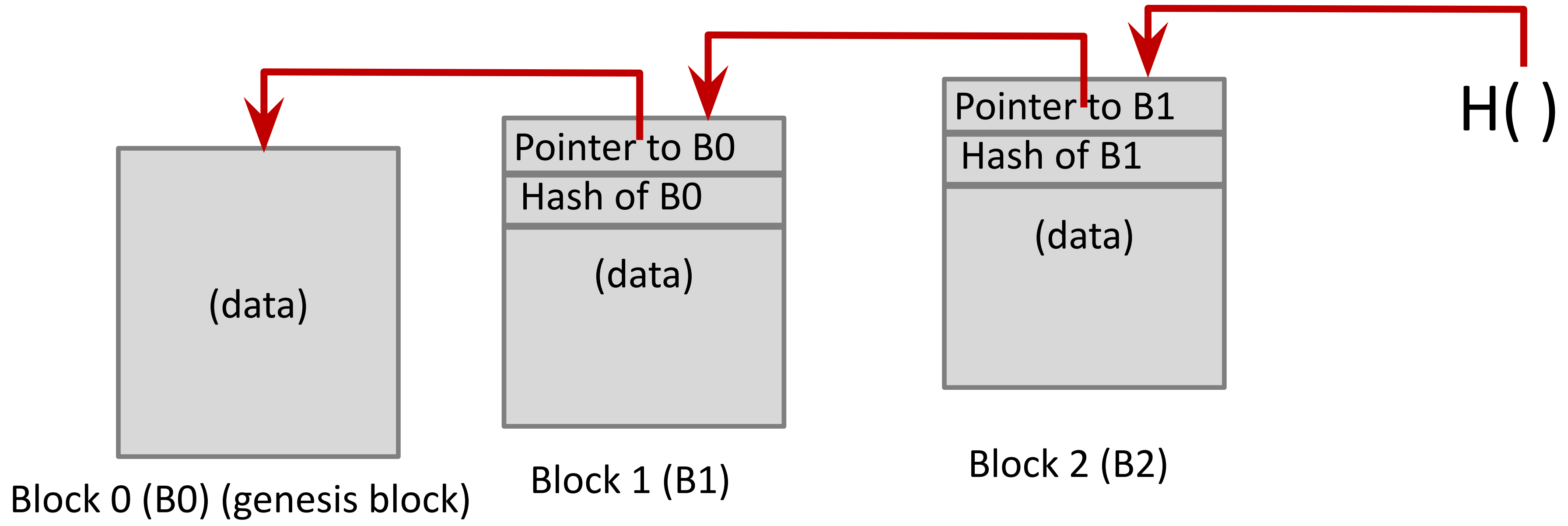
- Regular pointer: gives you a way to retrieve the information.
- Example: char *str = new char [10];
- Hash pointer: a pointer to where some info. is stored, together with a cryptographic hash of the info.
- If we have a hash pointer, we can ask to get the info back, and verify that it hasn't changed.



Data structures using hash pointers

- We can use hash pointers to build all kinds of data structures.
- We can take a familiar data structure that uses pointers such as a linked list or a binary search tree and implement it with hash pointers, instead of pointes as we normally would.
  - Linked list => Block chain
  - Sorted linked list => Hash list
  - Binary search tree => Hash tree / Merkle tree
- One can use hash pointers in any pointer-based data structure that has **no cycles.**

Block chain



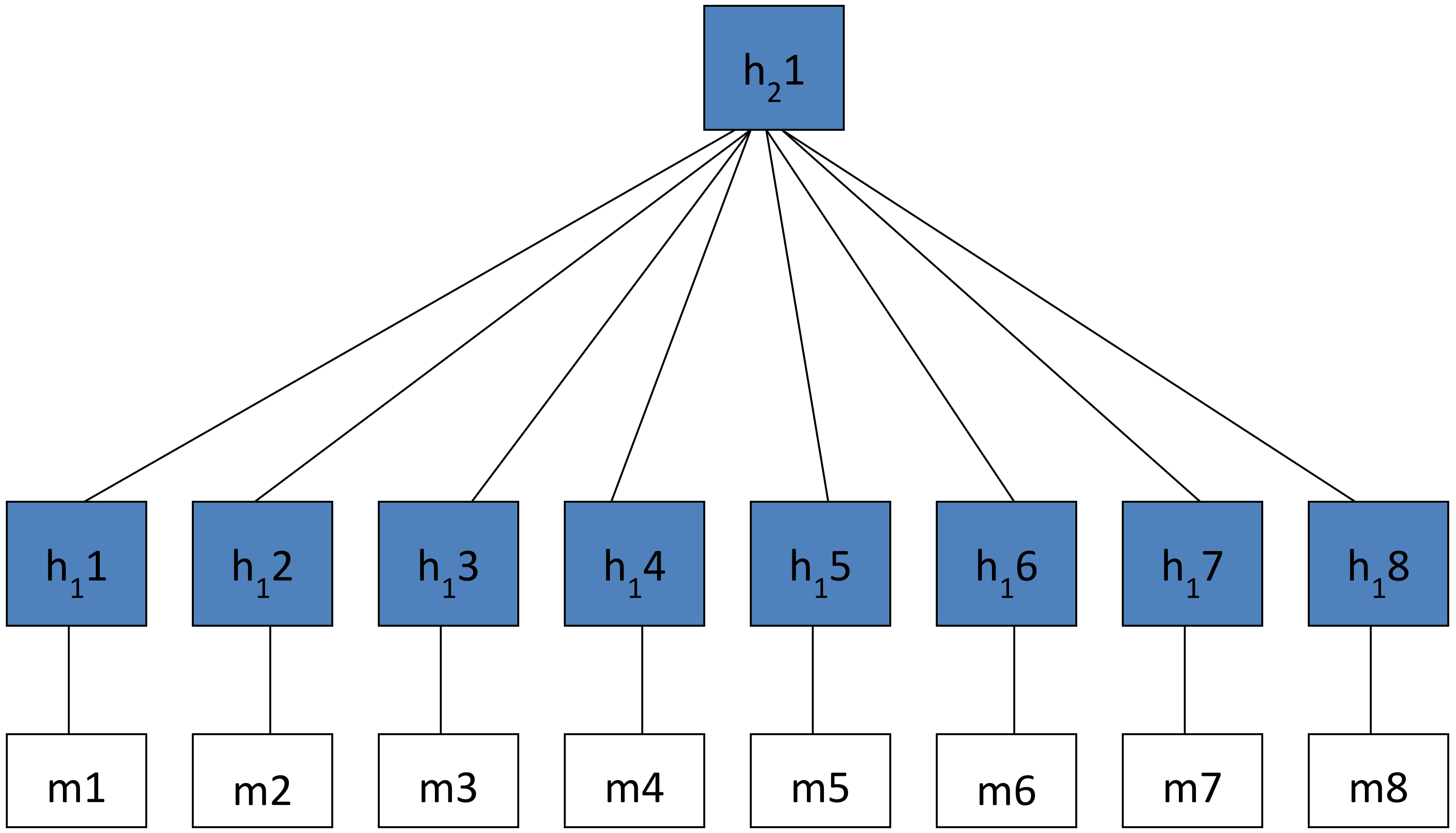
- In a block chain, the previous block pointer will be replaced with a hash pointer.
- So each block not only tells us where the value of the previous block was, but it also contains a digest of that value that allows us to verify that the value hasn't changed.
- If an adversary modifies data anywhere in the block chain, it will result in the hash pointer in the following block being incorrect.
- If we store the head of the list, then even if the adversary modifies all of the pointers to be consistent with the modified data, the head pointer will be incorrect, and we will detect the tampering.

Hash List

(taken from https://en.wikipedia.org/wiki/Hash_list and https://www.techopedia.com/definition/32921/hash-list)

- A *hash list* is a list of hashes (aka a set of hash values) related to sets of data blocks in a file, set of files in a folder system or some other connective array format
- Hash lists are used for many different purposes, such as fast table lookup (hash tables) and distributed databases (distributed hash tables).
- A hash list shows how a set of hash values are related: how they collectively work to store data from a given "block" or unified collection.
- A hash list is an extension of the concept of hashing an item (for instance, a file).
- A hash list is a subtree of a Merkle tree.

Hash List



Read more

- Read more at https://blockonomi.com/merkle-tree/

Text-search Data Structures

(slides taken from http://people.cs.aau.dk/~simas/aalg04/)

- Goal of the lecture:
  - ***Dictionary ADT*** *for strings:*
    - *to understand the principles of* ***tries****, compact tries, Patricia tries*

Dictionary ADT for Strings

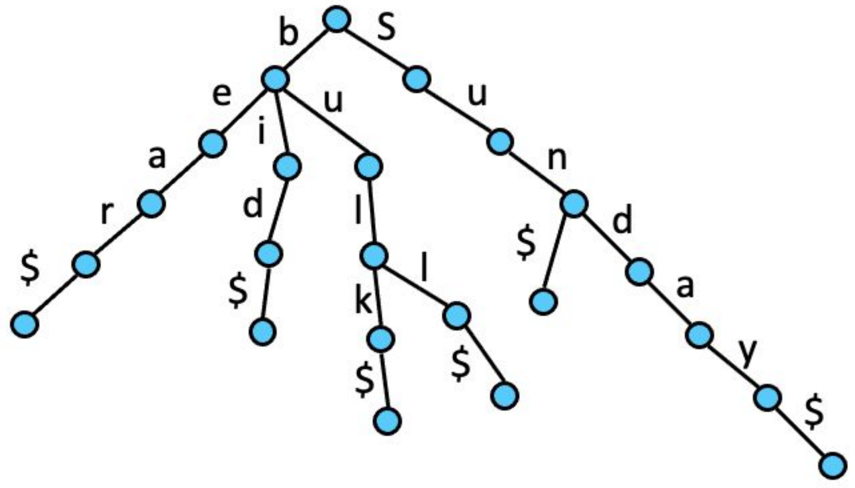
- Creating a dictionary ADT for strings
  - Elements: text strings
  - Operations:
    - *search*(*x*) – checks if string *x* is in the set
    - *insert*(x) – inserts a new string *x* into the set
    - *delete*(*x*) – deletes the string equal to *x* from the set of strings
- Assumptions, notation:
  - *n* strings, *N* characters in total
  - *m* – length of *x*
  - Size of the alphabet $d = |\Sigma|$

BST as dictionary for strings

- Using binary search trees has some issues:
  - Keys are of varying length
  - A lot of strings share similar prefixes (beginnings) – potential for saving space
  - Let’s count comparisons of characters.
    - What is the worst-case running time of searching for a string of length *m*?

Tries

- *Trie* – a data structure for storing a set of strings (name from the word “re*trie*val”):
  - Let’s assume, all strings end with “$” (not in Σ)



Set of strings: {**`bear`**, **`bid`**, **`bulk`**, **`bull`**, **`sun`**, **`Sunday`**}

Tries (cont.)

- Properties of a ***trie***:
  - A multi-way tree.
  - Each ***node*** has from 1 to *d* children.
  - Each ***edge*** of the tree is labeled with a character.
  - Each ***leaf*** node corresponds to the stored string, which is a concatenation of characters on a path from the root to this node.

Search and Insertion in Tries

```
Trie-Search(t, P[k..m])  //inserts string P into t
01 if t is leaf then return true
02 else if t.child(P[k])=nil then return false
03      else return Trie-Search(t.child(P[k]), P[k+1..m])
```

- The search algorithm just follows the path down the tree (starting with Trie-Search(*root*, *P*[0..*m*]))

```
Trie-Insert(t, P[k..m])
01 if t is not leaf then  //otherwise P is already present
02    if t.child(P[k])=nil then
03         Create a new child of t and a "branch" starting
                    with that chlid and storing P[k..m]
04    else Trie-Insert(t.child(P[k]), P[k+1..m])
```

- How would the delete work?

Trie Node Structure

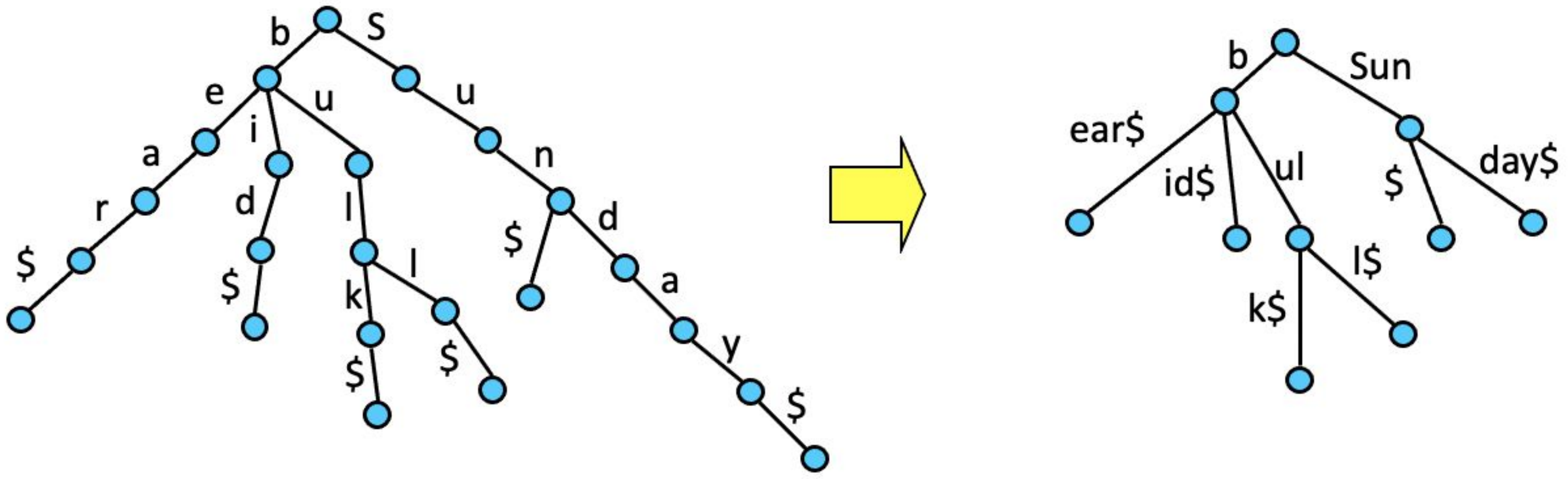
- "Implementation detail"
  - What is the node structure? = What is the complexity of the t.*child*(c) operation?:
    - An **array** of child pointers of size *d*: waste of space, but *child*(c) is *O*(1)
    - A **hash table** of child pointers: less waist of space, *child*(c) is expected *O*(1)
    - A **list** of child pointers: compact, but *child*(c) is *O*(*d*) in the worst-case
    - A **binary search tree** of child pointers: compact and *child*(c) is *O*(lg *d*) in the worst-case

Analysis of the Trie

- Size:
  - $O(N)$ in the worst-case
- Search, insertion, and deletion (string of length m):
  - depending on the node structure: $O(dm)$, $O(m \lg d)$, $O(m)$
  - Compare with the string BST
- Observation:
  - Having chains of one-child nodes is wasteful

Compact Tries

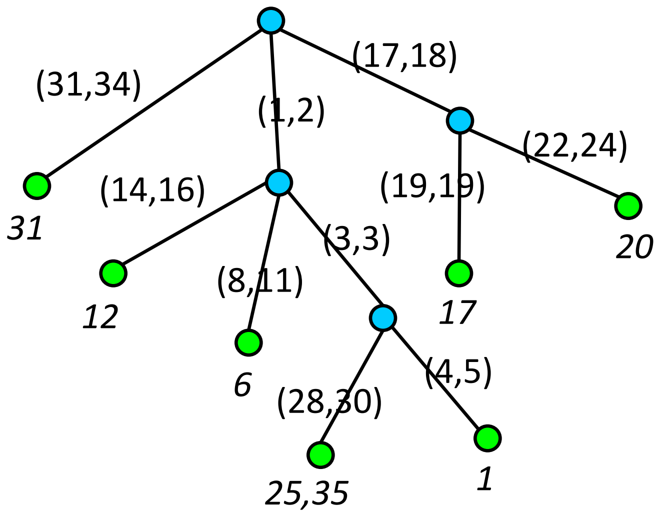
- *Compact Trie*:
  - Replace a *chain* of one-child nodes with an edge labeled with a string
  - Each non-leaf node (except root) has at least two children

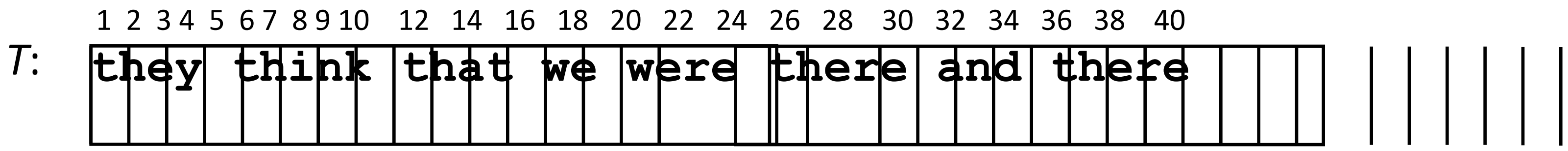


Compact Tries (contd.)

- Implementation:
  - Strings are external to the structure in one array, edges are labeled with indices in the array (*from*, *to*)
- Can be used to do *word matching*: find where the given word appears in the text.
  - Use the compact trie to “store” all words in the text
  - Each child in the compact trie has a list of indices in the text where the corresponding word appears.

Word Matching with Tries





- To find a word *P*:
  - At each node, follow edge (*i*,*j*), such that *P*[*i*..*j*] = T[i..j]
  - If there is no such edge, there is no *P* in *T*, otherwise, find all starting indices of *P* when a leaf is reached

Word Matching with Tries (contd.)

- Building of a compact trie for a given text:
  - How do you do that? Describe the compact trie insertion procedure
  - Running time: $O(N)$
- Complexity of word matching: $O(m)$
- What if the text is in external memory?
  - In the worst-case we do $O(m)$ I/O operations just to access single characters in the text – not efficient

Text-Search Problem

- Input:
  - *Text T* = "**`carrara`**"
  - *Pattern P* = "**`ar`**"
- Output:
  - All occurrences of *P* in *T*
- Reformulate the problem:
  - *Find all suffixes of T that has P as a prefix!*
  - We already saw how to do a *word prefix* query.

```
carrara
 arrara
  rrara
   rara
    ara
     ra
      a
```

Text-search Algorithms

(slides taken from http://people.cs.aau.dk/~simas/aalg04/)

- Goals of the lecture:
  - *Naive text-search algorithm and its analysis;*
  - ***Horspool** algorithm and its analysis.*
  - ***Boyer-Moore** algorithm and its analysis*
  - ***Rabin-Karp** algorithm and its analysis;*
  - ***Knuth-Morris-Pratt** algorithm ideas;*
  - *Comparison of the **advantages and disadvantages** of the different text-search algorithms.*

Text Search (or String Matching) Problem

- **Input**: Alphabet Σ and two strings T[0..n-1] and P[0..m-1], containing symbols from alphabet Σ
- **Output**: a set S containing all "shifts" $0 \leq s \leq n-m$ such that T[s..s+m-1] = P
- Example:
  Input: Σ = {a, b, …, z}, T[0..17]="to be or not to be", and P[0..1]="be"
  Output: S = [3, 16]
- Chapter 32 in the textbook

(taken from Levitin, page 105)

- String P is called the *pattern:*

$$\begin{array}{ccccccccccc}
t_0 & \dots & t_i & \dots & t_{i+j} & \dots & t_{i+m-1} & & \dots & t_{n-1} & \text{text } T \\
 & & \updownarrow & & \updownarrow & & \updownarrow & & & & \\
 & & p_0 & \dots & p_j & \dots & p_{m-1} & \text{pattern } P & & &
\end{array}$$

- A brute-force algorithm:
  - Until string is exhausted do the following:
    - Align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right In the latter case
    - If a match, add index to the list of indices S
    - If not, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text.
  - Note that the last position in the text that can still be a beginning of a matching substring is n − m (provided the text positions are indexed from 0 to n − 1).
  - Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

Simple Algorithm

- Uses brute force

```
ALGORITHM BruteForceStringMatch(T [0..n − 1], P [0..m − 1] )
//Implements brute-force string matching
//Input: An array T [0..n − 1] of n characters representing a text and
// an array P [0..m − 1] of m characters representing a pattern
//Output: The indices of the characters in the text that start a
// matching substring or [] (empty set) if the search is unsuccessful
S = [] // empty list
for i ← 0 to n − m do
  match ← True
  for j ← 0 to m-1 do
    if P [j ] ≠ T [i + j ] then
      match ← False
      break // exit the for loop
    endif
  endfor
  if j == m then
    S = S ○ i
  endif
endfor
Return S
```

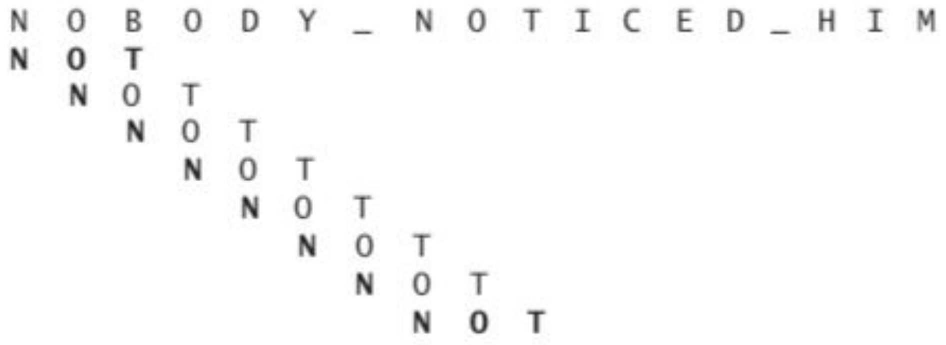


FIGURE 3.3 Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

Time complexity

- Worst case: O(nm)
- Average case (random text): O(n)
- Is it possible to obtain O(n) for any input?
  - Knutt-Morris-Pratt (1977): deterministic
  - Karp-Rabin (1981): randomized

Reverse naïve algorithm

- Why not search from the end of *P*?
  - Boyer and Moore

```
Reverse-Naive-Search(T,P)
for s ← 0 to n – m
   j ← m – 1   // start from the end
   // check if T[s..s+m–1] = P[0..m–1]
   while T[s+j] = P[j] do
      j ← j - 1
      if j < 0 return s
return –1
```

- Running time is exactly the same as of the naïve algorithm…

Occurrence heuristic

- Boyer and Moore added two heuristics to reverse naïve, to get an $O(n+m)$ algorithm, but it's complex
- Horspool suggested just to use the modified *occurrence* heuristic:
  - *After a mismatch, align* $T[s+m-1]$ *with the rightmost occurrence of that letter in the pattern* $P[0..m-2]$
  - Examples:
    - T= "**`detective date`**" and P= "**`date`**"
    - T= "**`tea kettle`**" and P= "**`kettle`**"

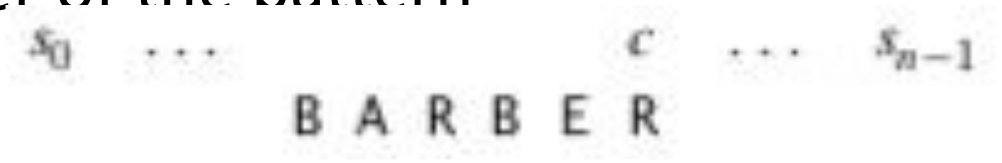
Input-enhancement Algorithms

- Input enhancement: preprocess the pattern to get some information about it, store this information in a table, and use that information during an actual search for the pattern in a given text
- The best-known algorithms:
  - Knutt-Morris-Pratt algorithm: compare characters of a pattern with their counterparts in a text from left to right
  - Boyer-Moore algorithm: compare characters of a pattern with their counterparts in a text from right to left
- Before we proceed with Boyer-Moore, we start with a simplified version of Horspool

Horspool’s Algorithm

(taken from Levitin, page 259)

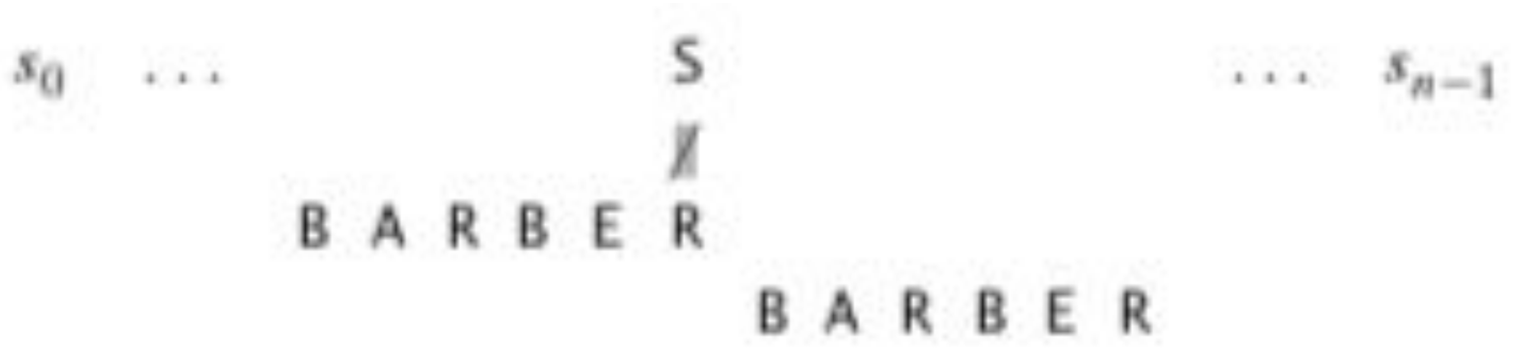
- When comparing a pattern against a position in the text, if a mismatch occurs, we need to shift the pattern to the right
- The shift would be as large as possible without risking the possibility of missing a matching substring in the text
- Idea: determine the size of the shift by looking at the character *c* of the text that is aligned against the last character of the pattern



  – Four cases:

Case 1

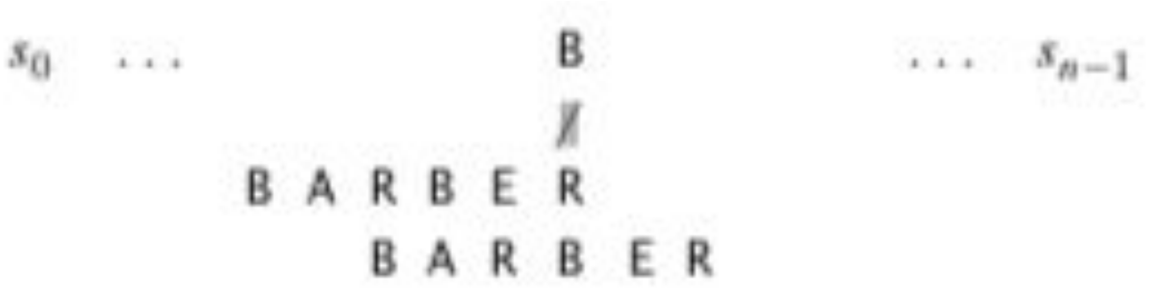
- Case 1: There are no *c*’s in the pattern; for example if *c* is S



  - Then we can safely shift the pattern by its entire length

Case 2

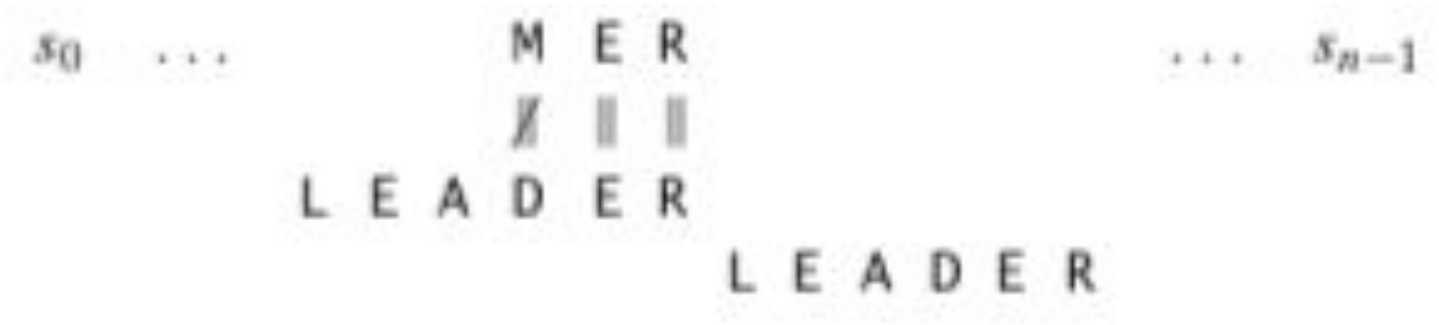
- Case 2: If there are occurrences of character *c* in the pattern but it is not the last one there;
  for example if *c* is letter B



  - Then the shift should align the rightmost occurrence of *c* in the pattern with the *c* in the text

Case 3

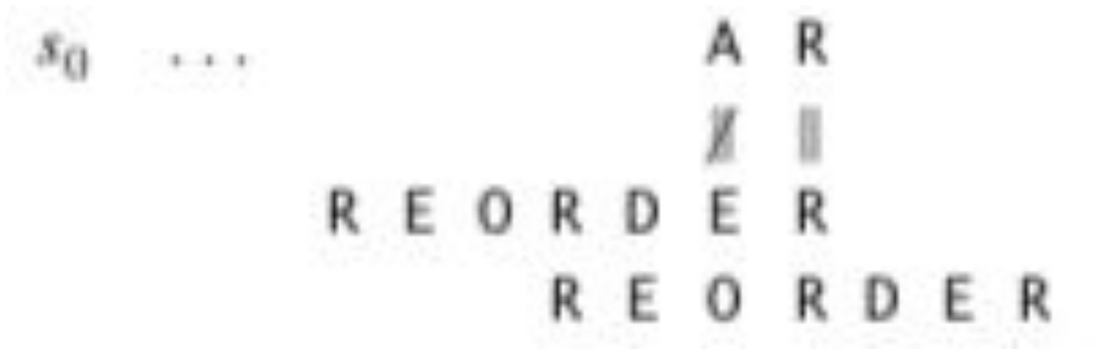
- Case 3: If *c* happens to be the last character in the pattern but there are no *c*'s among its other m − 1 characters; for example if *c* is letter R



  - Similar to Case 1: the pattern should be shifted by the entire pattern's length *m*

Case 4

- Case 4: If *c* happens to be the last character in the pattern and there are other *c*'s among its first m − 1 characters; for example if *c* is letter R



  – Similar to Case 2: the rightmost occurrence of *c* among the first m − 1 characters in the pattern should be aligned with the text's *c*

What we need to do

- We can precompute shift sizes and store them in a table called *shift table*
- The table will be indexed by all possible characters that can be encountered in a text, including, for natural language texts, the space, punctuation symbols, and other special characters.
- The table's entries will indicate the shift sizes computed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m & \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern} \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters of the pattern to its last character} & \text{otherwise} \end{cases}$$

Example of shift table

- Consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores).
- The shift table, as we mentioned, is filled as follows:

Character *c*	A	B	C	D	E	F	…	R	…	Z	_
Shift *t(c)*	4	2	6	6	1	6	6	3	6	6	6

Algorithm to construct the shift table

- Initialize all the entries to the pattern's length m and scan the pattern left to right repeating the following step $m - 1$ times: for the j th character of the pattern ($0 \le j \le m - 2$), overwrite its entry in the table with $m - 1 - j$, which is the character's distance to the last character of the pattern.
- Note that since the algorithm scans the pattern from left to right, the last overwrite will happen for the character's rightmost occurrence - exactly as we would like it to be.

```
ALGORITHM ShiftTable(P [0..m − 1] )
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P [0..m − 1] and an alphabet of possible characters
//Output: Table[0..size − 1] indexed by the alphabet's characters and
// filled with shift sizes computed by formula (7.1)
1: for i ← 0 to size − 1 do
2:   Table[i] ← m
3: for j ← 0 to m − 2 do
4:   Table[P [j ]] ← m − 1 − j
5: return Table
```

Example

- For the pattern BARBER, all the table’s entries will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively.

Horspool’s Algorithm: Steps

Step 1 For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched (then stop) or a mismatching pair is encountered. In the latter case, retrieve the entry t (c) from the c’s column of the shift table where c is the text’s character currently aligned against the last character of the pattern, and shift the pattern by t(c) characters to the right along the text.

Here is pseudocode of Horspool’s algorithm.

Horspool’s Algorithm: pseudocode

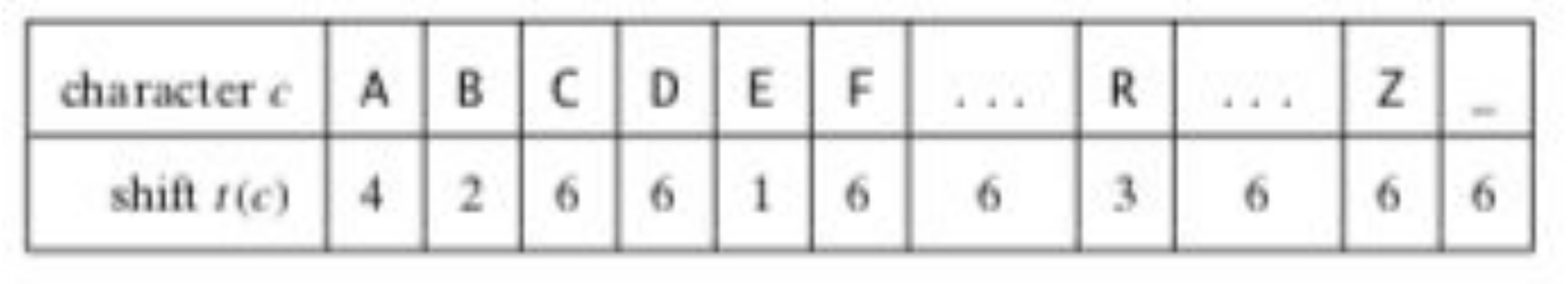
```
HorspoolMatching(P [0..m − 1], T [0..n − 1] )
//Implements Horspool’s algorithm for string matching
//Input: Pattern P [0..m − 1] and text T [0..n − 1]
//Output: The index of the left end of the first matching substring
// or −1 if there are no matches
ShiftTable(P [0..m − 1] )
1:  i = m-1
2:  while i <= n-1 do
3:      k = 0
4:      while k <= m-1 and P[m-1-k] == T[i-k] do
5:          k = k+1
6:          if k == m
7:              return i-m+1
8:      i = i + Table[T[i]]
9:  return -1
```

Time Complexity of Horspool’s algorithm

- The worst-case efficiency is in O(nm)
- For random texts, it is in Θ(n)
- Although in the same efficiency class, Horspool’s algorithm is obviously faster on average than the brute-force algorithm.

Example

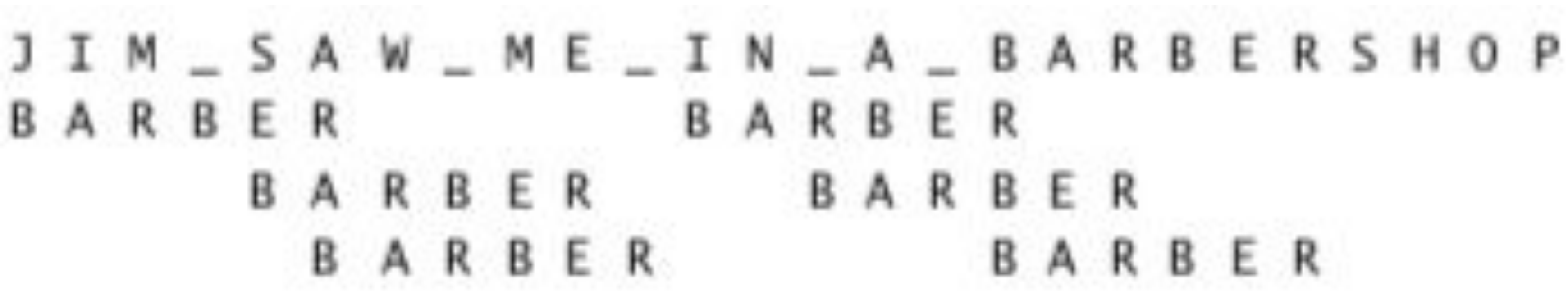
- Consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores).
- The shift table, as we mentioned, is filled as follows:



character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

Contd.

- The actual search in a particular text proceeds as follows:

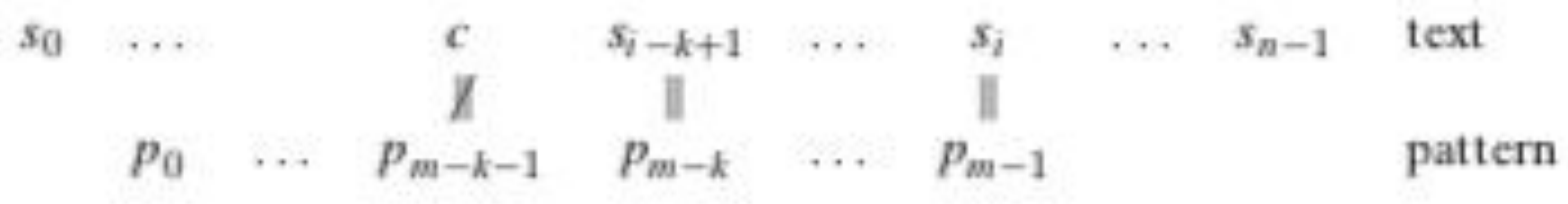


```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R           B A R B E R
        B A R B E R       B A R B E R
          B A R B E R           B A R B E R
```

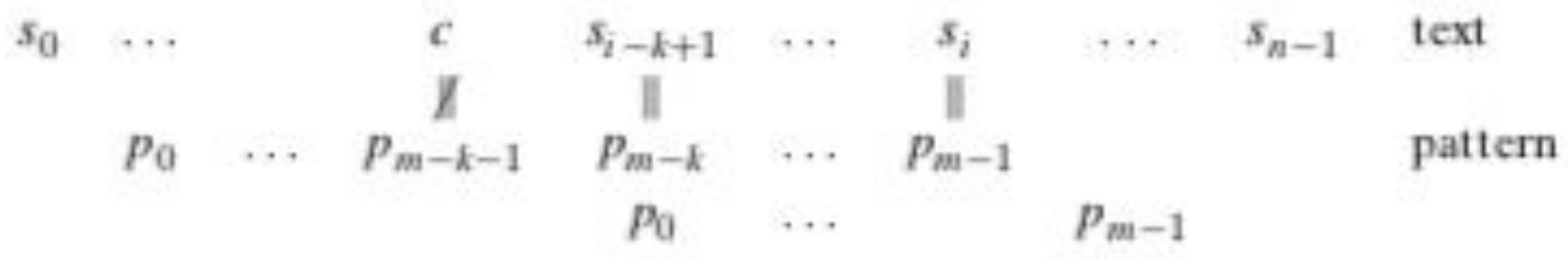
Boyer-Moore Algorithm

- If the first comparison of the rightmost character in the pattern with the corresponding character c in the text fails, the algorithm does exactly the same thing as Horspool's algorithm, namely, it shifts the pattern to the right by the number of characters retrieved from the shift table
- The two algorithms act differently, however, after some positive number k (0 < k < m) of the pattern's characters are matched successfully before a mismatch is encountered:

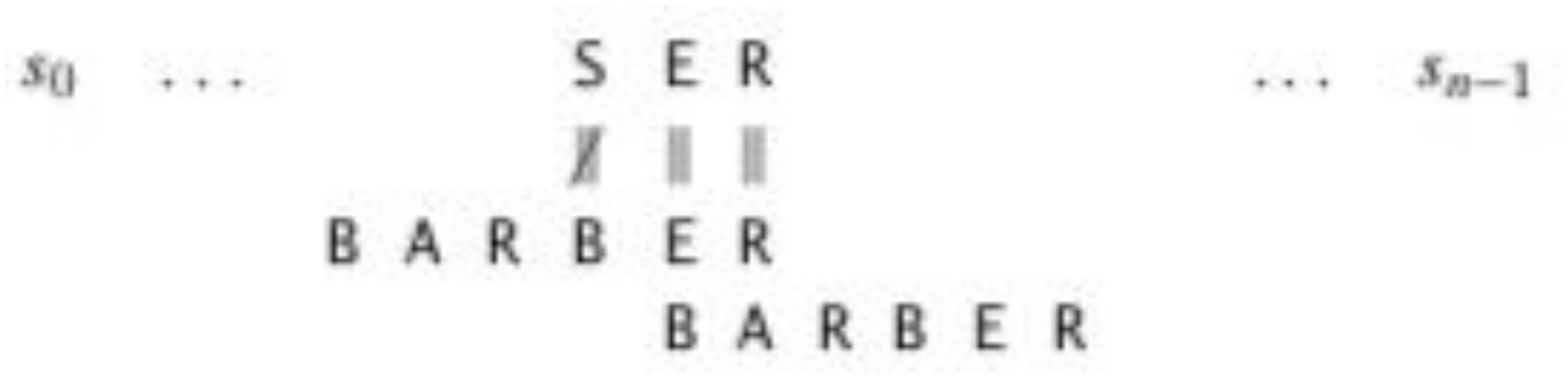
$$\begin{array}{ccccccccccc}
s_0 & \ldots & & & c & s_{i-k+1} & \ldots & s_i & \ldots & s_{n-1} & \text{text} \\
 & & & & \not\parallel & \parallel & & \parallel & & & \\
 & p_0 & \ldots & & p_{m-k-1} & p_{m-k} & \ldots & p_{m-1} & & & \text{pattern}
\end{array}$$


$s_0 \;\ldots\; c \;\; s_{i-k+1} \;\ldots\; s_i \;\ldots\; s_{n-1}$ text
$\ne \quad \| \quad \|$
$p_0 \;\ldots\; p_{m-k-1} \;\; p_{m-k} \;\ldots\; p_{m-1}$ pattern

- Determines the shift size by considering two quantities
- Bad-symbol shift: If *c* is not in the pattern, we shift the pattern to just pass this *c* in the text. Conveniently, the size of this shift can be computed by the formula $t_1(c) - k$ where $t_1(c)$ is the entry in the shift table


$s_0 \;\ldots\; c \;\; s_{i-k+1} \;\ldots\; s_i \;\ldots\; s_{n-1}$ text
$\ne \quad \| \quad \|$
$p_0 \;\ldots\; p_{m-k-1} \;\; p_{m-k} \;\ldots\; p_{m-1}$ pattern
$p_0 \;\ldots\; p_{m-1}$

- For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter S in the text, we can shift the pattern by $t_1(S) - 2 = 6 - 2 = 4$ positions:



- Good-suffix shift: a successful match of the last $k > 0$ characters of the pattern; we refer to the ending portion of the pattern as its *suffix* of size k and denote it suff (k). We now apply the reasoning that guided us in filling the bad-symbol shift table, which was based on a single alphabet character c, to the pattern’s suffixes of sizes $1, \ldots, m - 1$ to fill in the good-suffix shift table.
  - If there is another occurrence of suff(k) not preceded by the same character as in its rightmost occurrence, we can shift the pattern by the distance d_2 between such a second rightmost occurrence (not preceded by the same character as in the rightmost occurrence) of suff (k) and its rightmost occurrence.

- If there is no other occurrence of suff (k) not preceded by the same character as in its rightmost occurrence, we need to find the longest prefix of size $l < k$ that matches the suffix of the same size l. If such a prefix exists, the shift size d_2 is computed as the distance between this prefix and the corresponding suffix; otherwise, d_2 is set to the pattern’s length m.

Boyer-Moore Algorithm: Steps

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described earlier.

Step 3 Align the pattern against the beginning of the text.

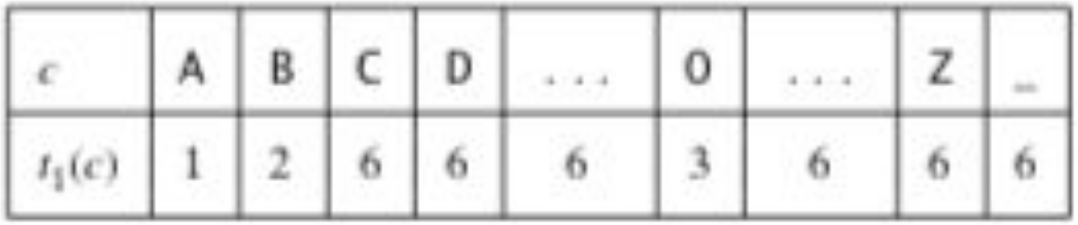
Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

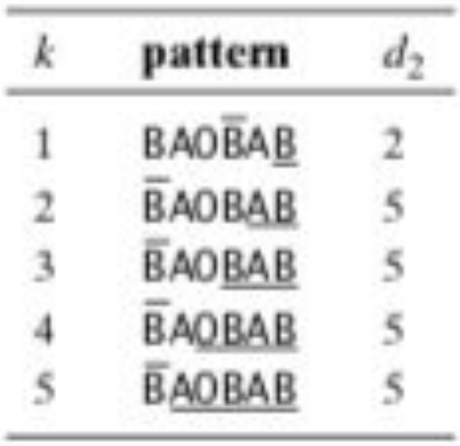
Example

- Searching for the pattern BAOBAB in a text made of English letters and spaces.
- The bad-symbol table looks as follows:



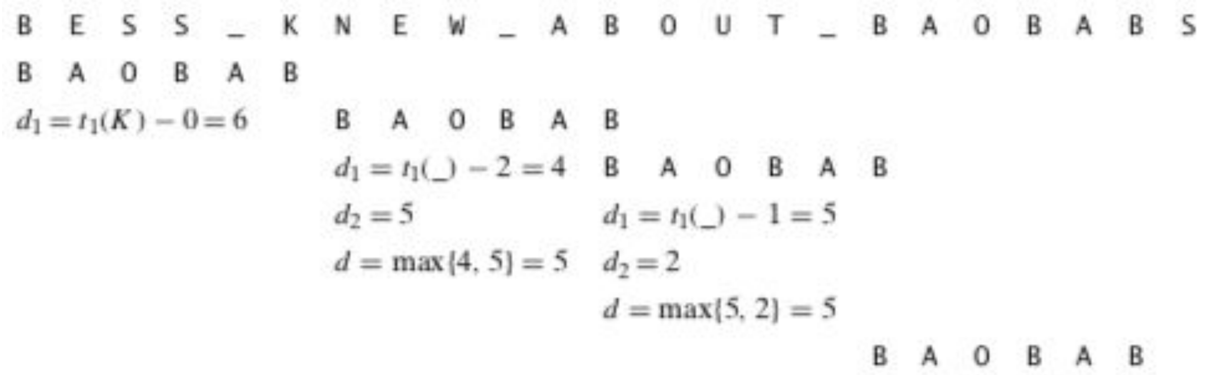
c	A	B	C	D	...	O	...	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

- The good-suffix table:



k	**pattern**	d_2
1	$\text{BAO}\overline{\text{B}}\text{A}\underline{\text{B}}$	2
2	$\overline{\text{B}}\text{AOB}\underline{\text{AB}}$	5
3	$\overline{\text{B}}\text{AO}\underline{\text{BAB}}$	5
4	$\overline{\text{B}}\text{A}\underline{\text{OBAB}}$	5
5	$\overline{\text{B}}\underline{\text{AOBAB}}$	5

- The actual search for this pattern in the text



```
B E S S _ K N E W _ A B O U T _ B A O B A B S
B A O B A B
d1 = t1(K) − 0 = 6   B A O B A B
                     d1 = t1(_) − 2 = 4   B A O B A B
                     d2 = 5               d1 = t1(_) − 1 = 5
                     d = max{4, 5} = 5    d2 = 2
                                          d = max{5, 2} = 5
                                                    B A O B A B
```

- After the last B of the pattern fails to match its counterpart K in the text, the algorithm retrieves t_1(K) = 6 from the bad-symbol table and shifts the pattern by d_1 = max{t1(K) − 0, 1} = 6 positions to the right.
- The new try successfully matches two pairs of characters.
- After the failure of the third comparison on the space character in the text, the algorithm retrieves t1( ) = 6 from the bad-symbol table and d2 = 5 from the good-suffix table to shift the pattern by max{d1, d2} = max{6 − 2, 5} = 5.
- Note that on this iteration it is the good-suffix rule that leads to a farther shift of the pattern.
- The next try successfully matches just one pair of B’s.
- After the failure of the next comparison on the space character in the text, the algorithm retrieves t_1( ) = 6 from the bad-symbol table and d_2 = 2 from the good-suffix table to shift the pattern by max{d1,d2} = max{6 − 1, 2} = 5.
- Note that on this iteration it is the bad-symbol rule that leads to a farther shift of the pattern. The next try finds a matching substring in the text after successfully matching all six characters of the pattern with their counterparts in the text.

Rabin-Karp algorithm

(https://users.cs.duke.edu/~reif/courses/alglectures/indyk.lectures/lecture_22.pdf)

- Uses hashing:
  - Hash all substrings T[0..m-1], T[1..m], …T[n-m..n-1]
  - Hash the pattern P[0..m-1]
  - Report the strings that hash to the same value as P
- Challenge: how to hash n-m substrings, each of length m, in O(n) time?

Converting strings to decimal values

(textbook, pg 990-991)

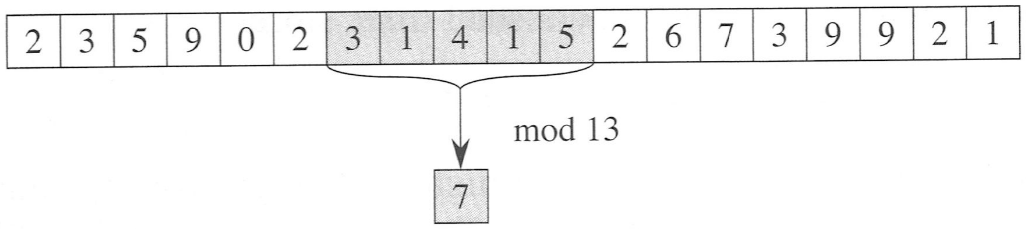
- For the pattern P[0..m-1] we need to produce a decimal value
- Consider Σ ={0,1,…9} and consider each character to be a digital digit; e.g. "31415" will have the decimal value of 31,415
  - Otherwise we use polynomial coding with a =|Σ| to convert form any string to a decimal value
- Let p be its decimal value of the pattern P[0..m-1] and let t_s be the decimal value of the substrings T[s..(m-1)+s]
- The decimal value p can be computed in O(m) time and all $t_0, t_1, \ldots t_{n-m}$ can be computed in O(n-m+1) time; why?
  - Because t_{s+1} can be computed from t_s in constant time, since $t_{s+1}=10(t_s-10^{m-1}T[s])+T[s+m]$ and computing 10^{m-1} can be done in O(log m)

- We might get large decimal values, so we apply hashing by taking modulo some q, where q is a prime such that 10q fits within one computer word
  - In general, we choose q to be a prime such that $q \cdot |\Sigma|$ fits in a computer word
- We need to compare p mod q with each $t_0, t_1, \ldots t_{n-m}$ where $t_{s+1} = (10(t_s - (10^{m-1} \bmod q)T[s]) + T[s+m]) \bmod q$
- Therefore we can find all occurrences of the pattern P[0..m-1] in the text T[0..n-1] with O(m) preprocessing time and O(n-m+1) matching time

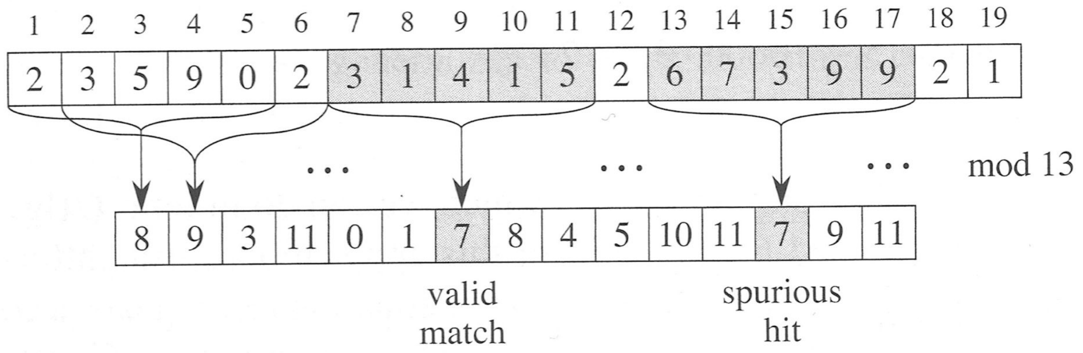
Rabin-Karp Algorithm

```
Rabin-Karp-Searcher(T,P)
q ← a prime larger than m
h ← $10^{m-1}$ mod q  // run a loop multiplying by 10 mod q
fp ← 0; ft ← 0
for i ← 0 to m-1  // preprocessing
   fp ← (10*fp + P[i]) mod q
   ft ← (10*ft + T[i]) mod q
for s ← 0 to n – m  // matching
   if fp = ft then   // run a loop to compare strings
      if P[0..m-1] = T[s..s+m-1] return s
   ft ← ( (ft – T[s]*h)*10 + T[s+m]) mod q
return –1
```

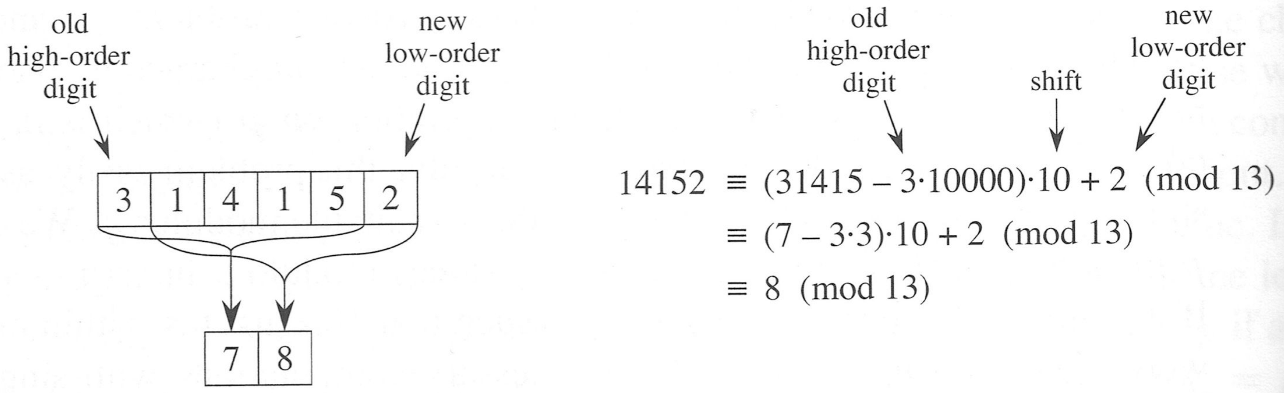
- How many character comparisons are done if *T* = "**2359023141526739921**" and *P* = "31415"? (see Fig. 32.5 on page 992 of the textbook)



(a)



(b)



$$
\begin{aligned}
14152 &\equiv (31415 - 3\cdot 10000)\cdot 10 + 2 \pmod{13} \\
&\equiv (7 - 3\cdot 3)\cdot 10 + 2 \pmod{13} \\
&\equiv 8 \pmod{13}
\end{aligned}
$$

(c)

Analysis

- If *q* is a prime, the hash function distributes *m*-digit strings evenly among the *q* values
  - Thus, only every *q*-th value of shift *s* will result in matching fingerprints (which will require comparing strings with *O*(*m*) comparisons)
- Expected running time (if *q* > *m*):
  - Preprocessing: *O*(*m*)
  - Outer loop: *O*(*n*-*m*)
  - All inner loops: $\frac{n-m}{q}m = O\left(n-m\right)$
  - Total time: *O*(*n*-*m*)
- Worst-case running time: *O*(*nm*)

Rabin-Karp in Practice

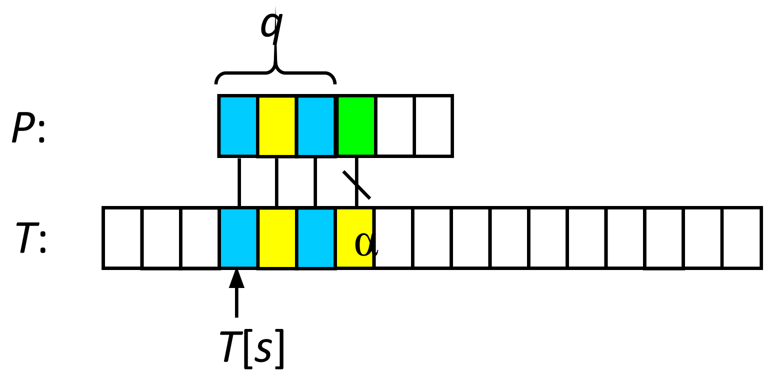
- If the alphabet has *d* characters, interpret characters as radix-*d* digits (replace 10 with *d* in the algorithm).
- Choosing prime $q > m$ can be done with randomized algorithms in O(*m*), or *q* can be fixed to be the largest prime so that 10**q* fits in a computer word.
- Rabin-Karp is simple and can be easily extended to two-dimensional pattern matching.

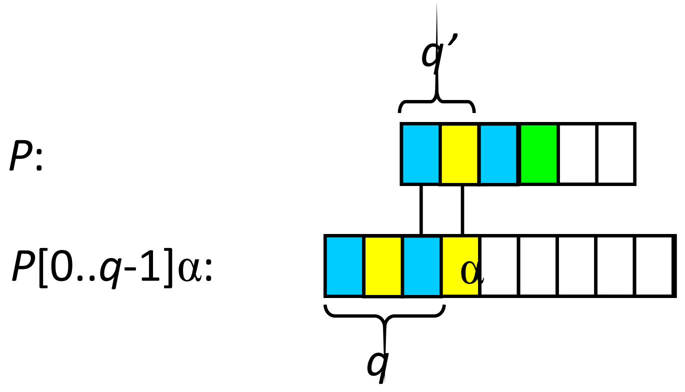
Searching in *n* comparisons

- The goal: each character of the text is compared only once!
- Problem with the naïve algorithm:
  - Forgets what was learned from a partial match!
  - Examples:
    - *T* = "**`Tweedledee and Tweedledum`**" and *P* = "**`Tweedledum`**"
    - *T* = "**`pappar`**" and *P* = "**`pappappappar`**"

General situation

- State of the algorithm:
  - Checking shift *s*,
  - *q* characters of *P* are matched,
  - we see a non-matching character α in *T*.
- Need to find:
  - Largest prefix "*P*-" such that it is a suffix of *P*[0..*q*-1]α:
    - New $q' = \max\{k \le q \mid P[0..k-1] = P[q-k+1..q-1]\alpha\}$





Greedy Algorithms

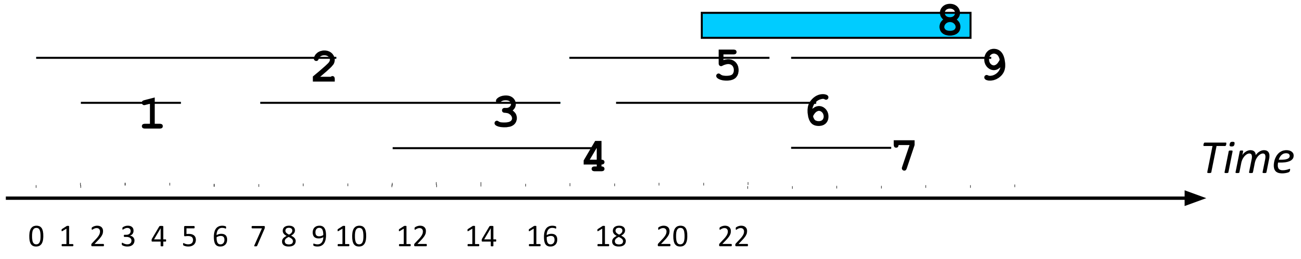
(slides taken from http://people.cs.aau.dk/~simas/aalg04/)

- Goals of the lecture:
  - *to understand the **principles** of the greedy algorithm design technique;*
  - *to understand the **example greedy algorithms** for activity selection and Huffman coding, to be able to **prove** that these algorithms find optimal solutions;*
  - *to be able to **apply** the greedy algorithm design technique.*

Activity-Selection Problem

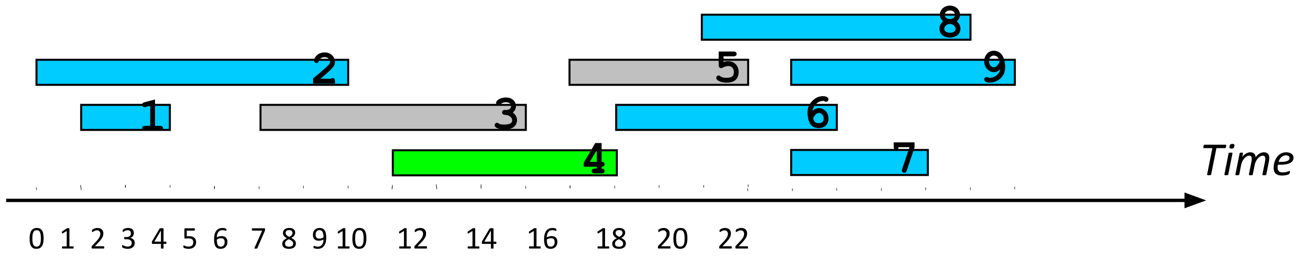
(textbook, pg 415-421)

- Description: scheduling several competing activities that require exclusive use of a common resource with the goal of selecting the maximum-size set of mutually compatible activities
- Input:
  - A set S of *n* activities, each with start and end times: *a*[*i*].*s* and *a*[*i*].*f*. The activity last during the interval [*a*[*i*].*s*, *a*[*i*].*f*)
- Output:
  - The ***largest*** subset of mutually *compatible* activities
    - Activities are compatible if their intervals do not intersect



“Straightforward” solution

- Let’s just pick (schedule) one activity *a*[*k*]
  - This generates two sets of activities compatible with it: *Before*(*k*), *After*(*k*)
    - E.g., Before(4) = {1, 2}; After(4) = {6,7,8,9}



  - Solution: let $S_{i,j}$ be the set of activities that start after activity a_i finishes and that finish before activity a_j starts

$$c[i,j] = \begin{cases} 0 & if\ S_{i,j} = \emptyset \\ \max\limits_{a_k \in S_{i,j}} \{c[i,k] + c[k,j] + 1\} & if\ S_{i,j} \neq \emptyset \end{cases}$$

Greedy choice

- What if we could choose "the best" activity (as of now) and be sure that it belongs to an optimal solution
  - We wouldn't have to check out all these sub-problems and consider all currently possible choices!
- Idea: Choose the activity that finishes first!
  - Then, solve the problem for the remaining compatible activities
  - Recursive function, RAS(..) that starts with RAS(a[1..n],0,n)

```
Recursive-Activity-Selector(a[1..n],k,n)    //returns
  a set of activities
01 m ← k + 1
02 while m ≤ n and a[m].s < a[k].f do
03    m ← m + 1
04 if m ≤ n then return {a[m]} ∪ RAS(a, m, n)
05          else return ∅
```

Greedy-choice property

- What is the running time of this algorithm?
- Does it find an optimal solution?:
  - We have to prove the *greedy-choice property*, i.e., that our locally optimal choice belongs to some globally optimal solution.
  - We have to prove the *optimal sub-structure* property (we did that already)
- The challenge is to choose the right interpretation of “the best choice”:
  - How about the activity that starts first
    - Show a *counter-example*

Huffman codes (textbook, pages 428-435)

- Compress data very effectively: savings 20-90% are typical
- Huffman's greedy alg uses a table with the frequency of each character to build an optimal way to represent each character as a binary string
- Let N be the number of distinct characters in the text
- Each character is represented by a unique binary string, which we call a *codeword* or *code*
  - *Fixed-length code* requires $\lceil \log_2 N \rceil$ bits
  - *Variable-length code* gives infrequent characters longer codes and frequent character shorter codes

Data Compression

- *Data compression* problem – strings *S* and *S'*:
  - *S -> S' -> S,* such that *|S'|<|S|*
- Text compression by coding with *variable-length* code:
  - Obvious idea – assign short codes to frequent characters: **"fabcadef"**

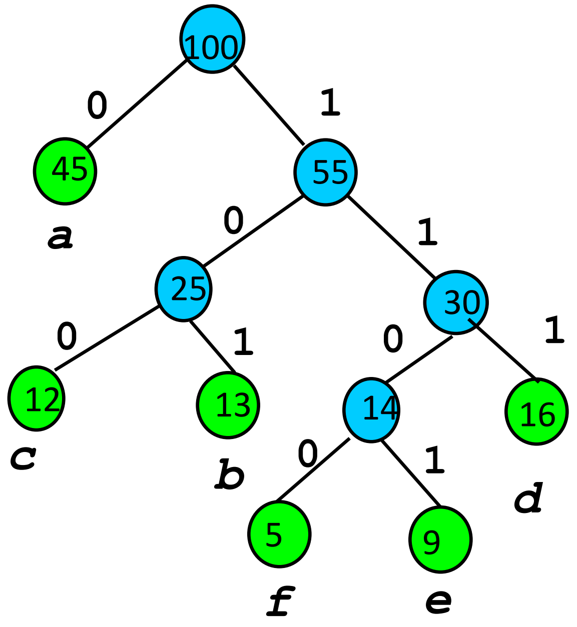
Frequency table: a 100K-character file, frequency given in order of 1K appearances

	a	**b**	**c**	**d**	**e**	**f**
Frequency	45	13	12	16	9	5
Fixed-length code	000	001	010	011	100	101
Variable-length code	0	101	100	111	1101	1100

  - How much do we save for each type? (300K bits for f.-l. and 224k for v.-l)

Prefix code

- Optimal code for given frequencies:
  - Achieves the minimal length of the coded text
- *Prefix code*: no codeword is prefix of another
  - It can be shown that optimal coding can be done with prefix code



- We can store all codewords in a *binary trie* – very easy to decode
  - Coded characters in leaves
  - Each node contains the sum of the frequencies of all descendants

Optimal Code/Trie

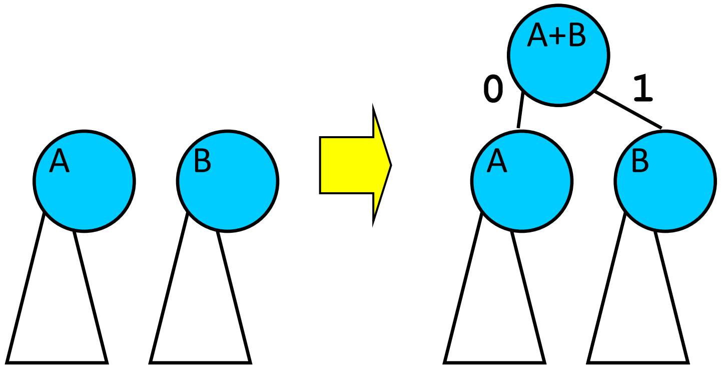
- The *cost* of the coding trie *T*:

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

  - *C* – the alphabet,
  - *f*(*c*) – frequency of character *c,*
  - $d_T(c)$ – depth of *c* in the trie (length of code in bits)
- Optimal trie – the one that minimizes *B*(*T*)
- Observation – optimal trie is always full:
  - Every non-leaf node has two children. Why?

Huffman Algorithm - Idea

- Huffman algorithm, builds the code trie bottom up. Consider a forest of trees:
  - Initially – one separate node for each character.
  - In each step – join two trees into a larger tree

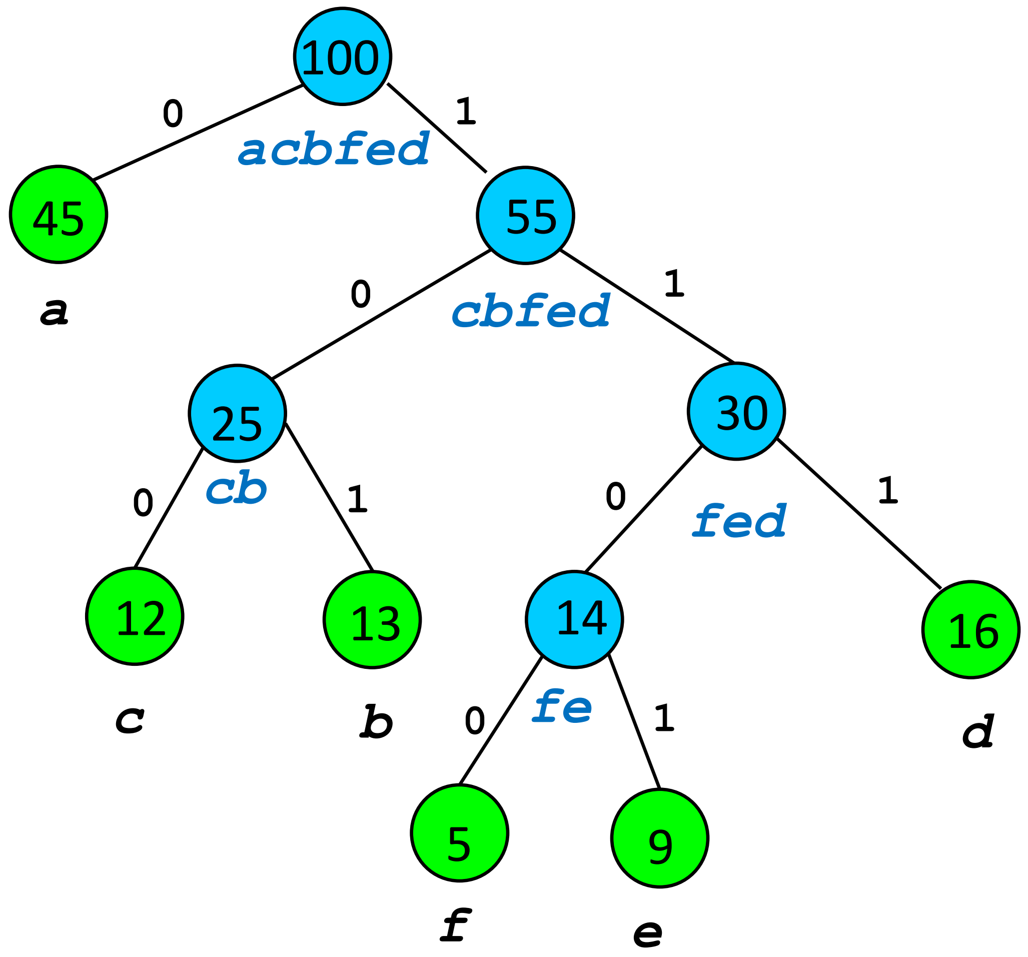


- Repeat this until one tree (trie) remains.
- Which trees to join? Greedy choice – the trees with the **smallest** frequencies!

Huffman Algorithm

```
Huffman(C)
Q.build(C) // Builds a min-priority queue on frequences
for i ← 1 to n−1 do
   Allocate new node z
   x ← Q.extractMin()
   y ← Q.extractMin()
   z.setLeft(x)
   z.setRight(y)
   z.setF(x.f() + y.f())
   Q.insert(z)
return Q.extractMin() // Return the root of the trie
```

- What is its running time?
- Run the algorithm on: "`oho ho, Ole`"


100
acbfed
0
1
45
a
55
cbfed
0
1
25
cb
0
1
12
c
13
b
30
fed
0
1
14
fe
0
1
5
f
9
e
16
d

Work out another example

- *A*ssign variable-length codes to characters in **“abracadabra”**
- First, compute the frequency table
- Compute the number of bits needed for fixed-length code: $3 = \lceil \log_2(5) \rceil$
- Now apply Huffman coding to compute the variable-length codes for each character

Frequency table

	a	**b**	**c**	**d**	**r**
Frequency	5	2	1	1	2
Fixed-length code	000	001	010	011	100
Variable-length code	**?**	**?**	**?**	**?**	**?**

Elements of Greedy Algorithms

- Greedy algorithms are used for optimization problems
  - Several choices must be made to arrive at an optimal solution
  - At each step, make the “locally best” choice, without considering all possible choices and solutions to sub-problems induced by these choices (compare to dynamic programming)
  - After the choice, only one sub-problem remains (smaller than the original)
- Greedy algorithms usually sort or use priority queues

Algorithm Framework

(taken from https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/07-max-flow.pdf)

- *Algorithm framework*: an algorithm with modular parts that can be swapped in for different performance properties; or to solve different but related problems
- Example: hash tables are a framework, can swap in
  - different collision resolution strategy (chaining, probing)
  - different hash function (universal hash, linear congruential hash, etc.)
- A framework generalizes several algorithm ideas into one pattern; "chunking"

Iterative Pattern

(taken from https://github.com/kevinwortman/advanced-algorithms-slides/blob/master/07-max-flow.pdf)

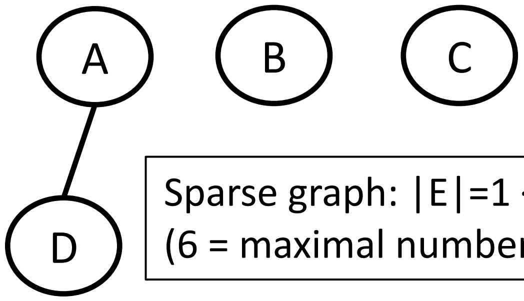
Recall greedy pattern:	Iterative pattern (a.k.a. fixed-point algorithm)
1. Initialize base-case result (variable) 2. For each piece of input (variable todo), update result	1. Initialize base-case result (variable) 2. While (result is not optimal): 2.1 Improve result in one step

- The fixed point is when the result becomes optimal
- Both greedy pattern and iterative pattern use a greedy heuristic: make a greedy choice to be executed in the next step

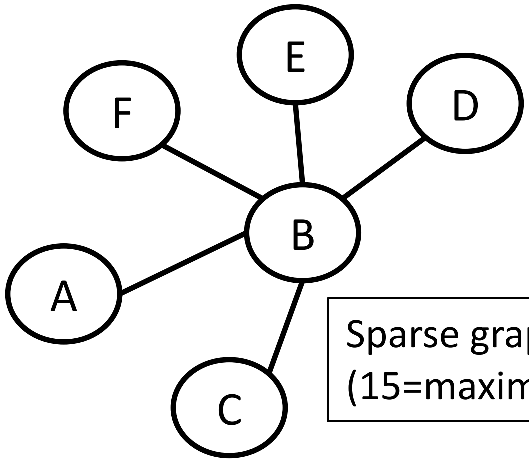
Review SSSP

Sparse versus Dense Graphs

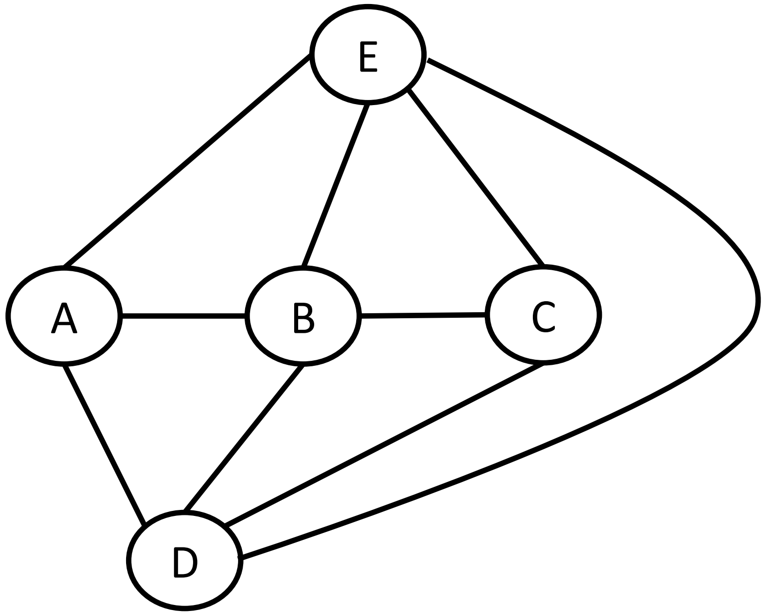
- Sparse graph: a graph in which the number of edges is close to the minimal number of edges (0)
- Examples
- Dense graph: a dense graph is a graph in which the number of edges is close to the maximal number of edges $\left(\frac{n \cdot (n-1)}{2}\right)$
- Examples on next slides



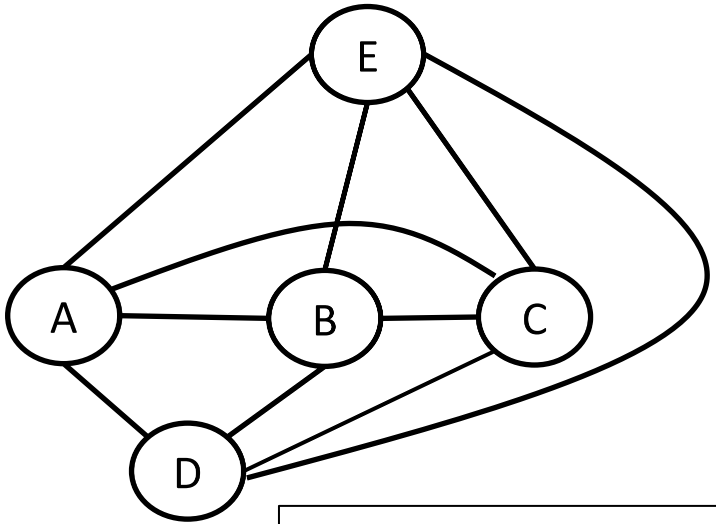
Sparse graph: |E|=1 < 6
(6 = maximal number of edges)



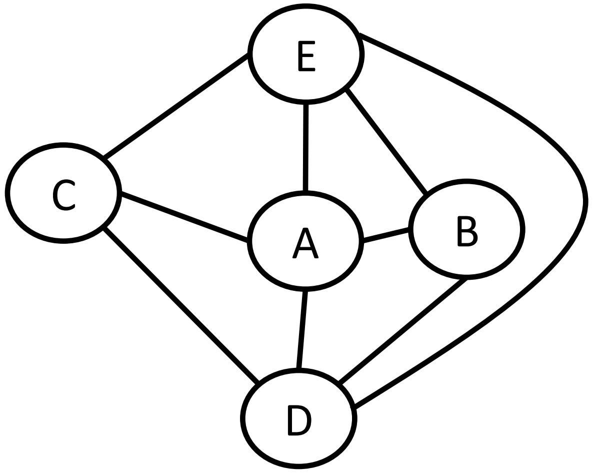
Sparse graph: |E|=5 < 15
(15=maximal number of edges)



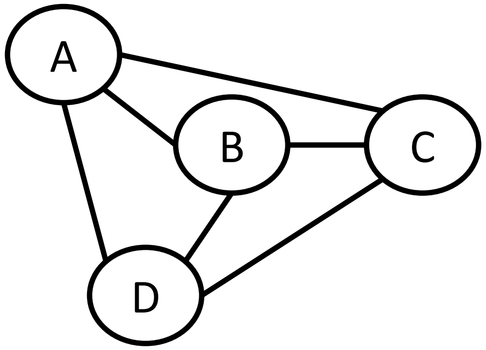
Dense graph: |E|=9 (close to
10=maximal number of edges)



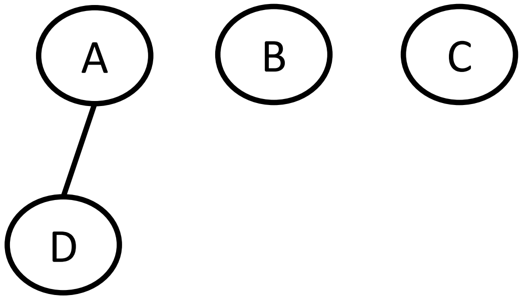
Dense graph: |E|=10 (close to
10=maximal number of edges)



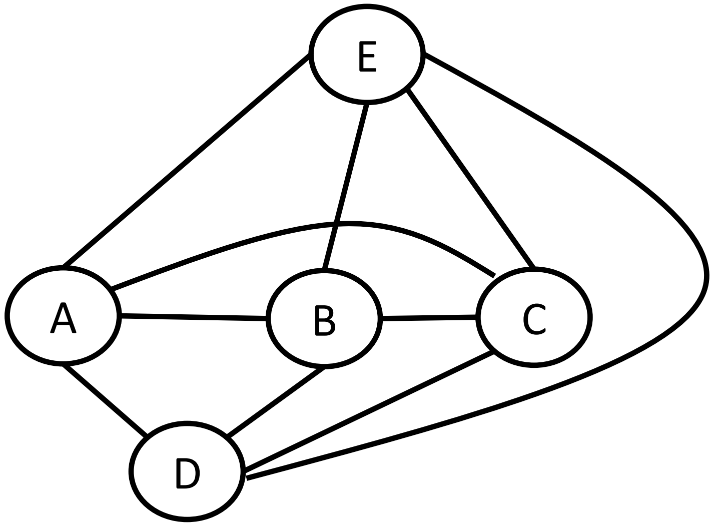
Dense



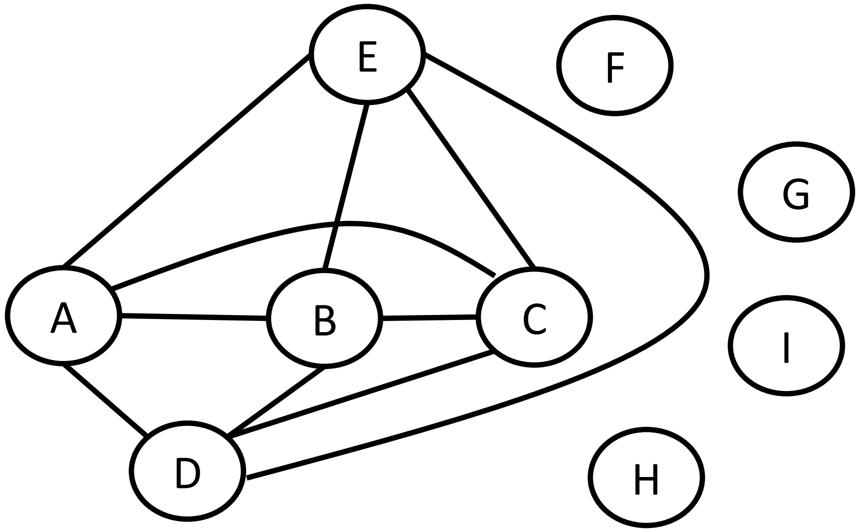
dense



sparse



dense



sparse

Single-source Shortest Path

- *Shortest path problems* involve computing minimal-weight paths between vertices in weighted graphs.
- In a weighted graph, each edge may have a unit cost, or each may have a numeric weight.
- Numeric weights may be unconstrained or be required to be non-negative.
- *Single source shortest path problems* involve computing paths originating from a designated *source* (or *start*) vertex
- *All-pairs shortest path problems* involve computing paths between all pairs of distinct vertices.

- $G = (V, E)$ a weighted undirected graph, V be the set of graph vertices, E be the set of graph edges, (an edge is a set of exactly two vertices)
- a *path* is any non-empty sequence of vertices $\langle p_0, p_1, \cdots p_{k-1} \rangle$ such that each $p_i \in V$ and every pair of adjacent vertices p_i, p_{i+1} is connected in G, so $\{p_i, p_{i+1}\} \in E$
- w_e is the weight of any edge e $\in E$
- $W(X)$ is the total weight of a path X

$$\mathrm{W}\langle p_0, p_1, \cdots p_{k-1} \rangle = \sum_{i=0}^{k-1} w_{\{p_i, p_{i+1}\}}$$

- $L_{s,v} = \langle s, \cdots v \rangle$ is a shortest path from start vertex $s \in V$ to end vertex v $\in V$
- $d_{s,v}$ is the shortest distance between s and v, i.e. the total weight of all edges visited by some $L_{s,v}$
- $p_{s,v}$ be the penultimate (next to last) vertex on a shortest path from s and v, i.e. $L_{s,v} = \left\langle s, \cdots, p_{s,v}, v \right\rangle$

- For a given G, s, v
  - $L_{s,v}$ and $d_{s,v}$ are only defined when s and v are connected
  - $p_{s,v}$ only exists when s and v are connected by a shortest path that visits at least 2 vertices
- We consider the single source shortest paths problem where edge weights are non-negative numbers (in fact positive numbers)
- Examples: driving directions, communication time in a network of routers

- The *non-negative single source shortest paths* problem is:

input: an undirected graph $G = (V, E)$ with non-negative edge weights, and a start vertex $s \in V$

output: two lists `distance` and `penultimate`, each of length exactly n, such that, for any $v \in V$

$$distance[v] = \begin{cases} d_{s,v} & \text{if } s \text{ and } v \text{ are connected, or} \\ \infty & \text{otherwise} \end{cases}$$

and

$$penultimate[v] = \begin{cases} p_{s,v} & \text{if } s \text{ and } v \text{ are connected, or} \\ s & \text{if } s = v\text{, or} \\ \infty & \text{if } s \text{ and } v \text{ are not connected} \end{cases}$$

- The list $penultimate$ encodes the identity of the vertices along any shortest path originating from s
- If we need a shortest path from s to a particular destination vertex v, $penultimate[v]$ is the second-to-last vertex on the shortest path from s to v, $penultimate[penultimate[v]]$ is the third-to-last vertex, $penultimate\left[penultimate[penultimate[v]]\right]$ is the fourth-to-last, and so on, until we get back to s

Dijkstra’s algorithm

- An efficient greedy algorithm for the non-negative single source shortest paths problem
- It does not work for graphs that have cycles of negative weight (example later)
- It maintains a partition $V = S \cup U$ of the vertex set V into two disjoint and nonempty parts, S and U; the set S is the set of *seen* vertices (explored so far), i.e. the algorithm has computed correct shortest paths from s to any vertex in S, and the set U is the set of *unseen* vertices (unexplored yet)
- The algorithm repeatedly picks an unseen vertex u from U according to a greedy heuristic, computes a valid shortest path from s to u, updates its data structures, and moves u from U to S accordingly
- Instead of using two lists S and U, we use a Boolean array `seen` such that $seen[v] = false$ is the vertex v is unseen, $seen[v] = \ true$ is the vertex v is seen

- Initially, all the vertices are unseen thus the array `seen` has all elements `false`, the array `distance` has all elements ∞, and the array `penultimate` has all elements equal to some value

  seen[ ] = false

  distance[ ] = ∞

  penultimate[ ] = nil (or some other value)

- Then the start vertex s becomes seen and update the values for s in the data structures:

  distance[s] = 0

  penultimate[s] = s

  seen[s] = True

Greedy step

- Then the algorithm repeatedly finds some edge $b = \{v, u\}$ such that v is seen and u is unseen and $\langle s, \cdots, v, u \rangle$ is a shortest path from s to u

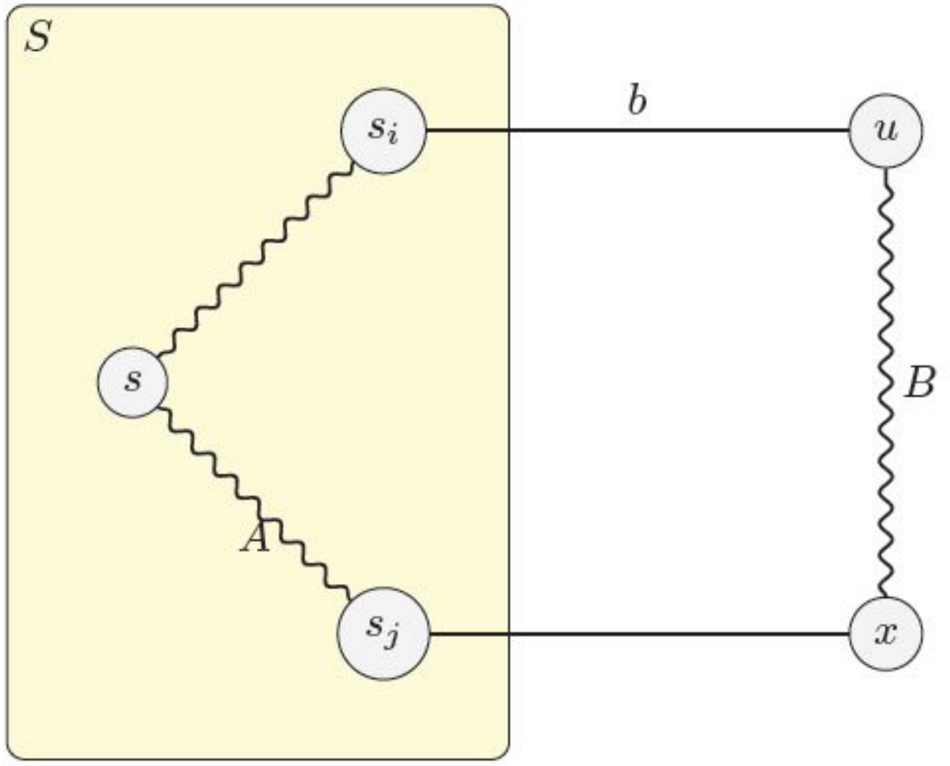
- If such a edge b does not exist, then terminate and return the data structures `distance` and `penultimate`
- Else, update its data structures:

  distance[u] = distance[v] + b.weight

  penultimate[u] = v

  seen[u] = True

- If `seen[u]` is True then `distance[u]` stores the correct value of $d_{s,u}$ and `penultimate[u]` stores the correct value of $p_{s,u}$
- Greedy step: choose the bridge edge $b = \{s_i, u\}$ that minimizes the value of $d_{s,s_i} + w(b)$



- Greedy step in details:
- Check all the edges that have one seen and one unseen endpoint
- Let $e = \{x, y\}$ be such an edge, with x the seen endpoint and y the unseen endpoint
- Calculate the distance from s to y using e as the last leg of the path
- Calculate all such distances from s to some unseen node y and the store the edge $e = \{x, y\}$ for which the distance is the smallest: use a variable si to store x and another variable u to store y
- In fact we found the bridge edge $b = \{s_i, u\}$ that minimizes the value of $d_{s,s_i} + w(b)$
- If such a bridge edge still exists, then we found a shortest path from s to u, so we add si as the parent of u along this shortest path (`penultimate[u] ← si`) and mark u as seen

```
while (true) {
  ● least_weight = ∞
    for all edges $e = \{x, y\} \in E$
        if seen[x] = true and seen [y]=false then {
            // found a bridge edge e between $S$ and $U$
            temp  = distance [x] + weight($\{x, y\}$)
            if (least_weight = ∞ or least_weight > temp) then {
                // found a path from s to y through x of smaller weight
                least_weight = temp
                // store the edge $e = \{x, y\}$
                si = x
                u = y
            }
        }
    }
    if (least_weight = ∞ ) then
    // all vertices are seen so break the while loop
    break
    else {
        // mark u as seen and update data structures
        distance [u] = least_weight
        penultimate[u] = si
        seen[u] = true
    }
}
```

```
seen[ ] = false
distance[ ] = ∞
penultimate[ ] = nil (or some other value)

distance[s] = 0
penultimate[s] = s
seen[s] = True

while (true) {
    least_weight = ∞
    for all edges $e = \{x, y\} \in E$
        if seen[x] = true and seen [y]=false then {
            // found a bridge edge e between $S$ and $U$
            temp  = distance [x] + weight($\{x, y\}$)
            if (least_weight = ∞ or least_weight > temp) then {
                // found a path from s to y through x of smaller weight
                least_weight = temp
                // store the edge $e = \{x, y\}$
                si = x
                u = y
            }
        }
    }
    if (least_weight = ∞ ) then
    // all vertices are seen so break the while loop
    break
    else {
        // mark u as seen and update data structures
        distance [u] = least_weight
        penultimate[u] = si
        seen[u] = true
    }
}
```

Algorithm Dijkstra_least_weight

- This version of Dijkstra’s algorithm runs in $O(|V| \cdot |E|)$ time
- The asymptotic running time can be reduced further

Dijkstra’s algorithm with min-priority queue

- Use a priority queue to greedily select the closest vertex not yet processed and perform the relaxation process on all its outgoing edges
- When we calculate the distance from s to y using e as the last leg of the path, if we found a value smaller than the currently stored value in `distance[y]`, then we can consider the newly found path a better alternative than the current path from s to y

```
**Procedure** Relax(x,y, distance, penultimate)
if (distance[y] = ∞ or distance[y] > distance [x] + weight({x,y}))
then
  distance [y] = distance [x] + weight({x,y})
  penultimate [y] = x
```

```
Initialize(distance, penultimate, s)
seen [ ] = false
Q = V // Q is a min-priority queue
while Q ≠ Ø {
    u = Extract-Min(Q)
    seen [ u ] = true
    for each vertex v adjacent to u
        Relax (u, v, distance, penultimate)
}
```

See next slide

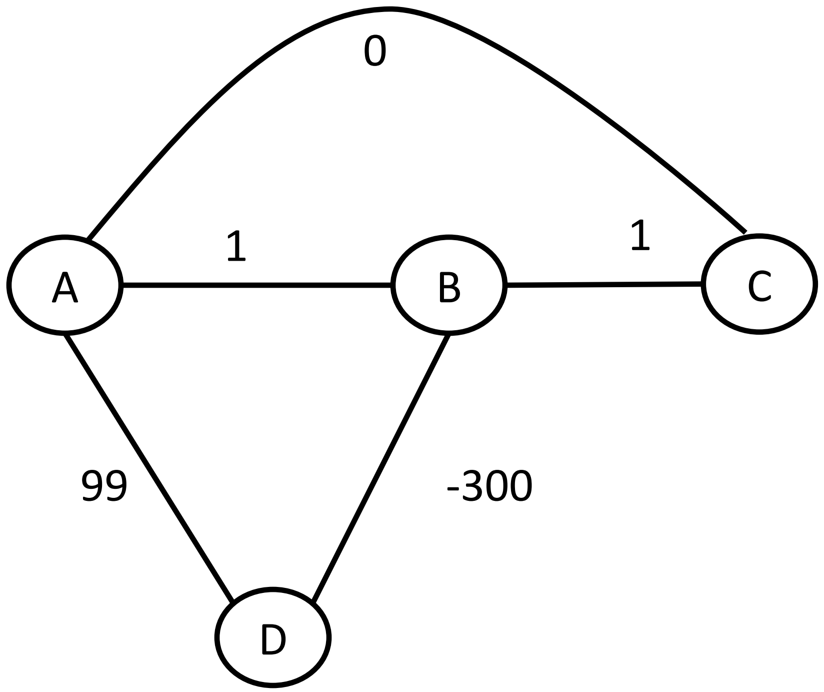
```
**Procedure** Initialize(distance, penultimate, s)
distance[ ] = ∞
penultimate[ ] = nil (or some other value)

distance[s] = 0
penultimate[s] = s
```

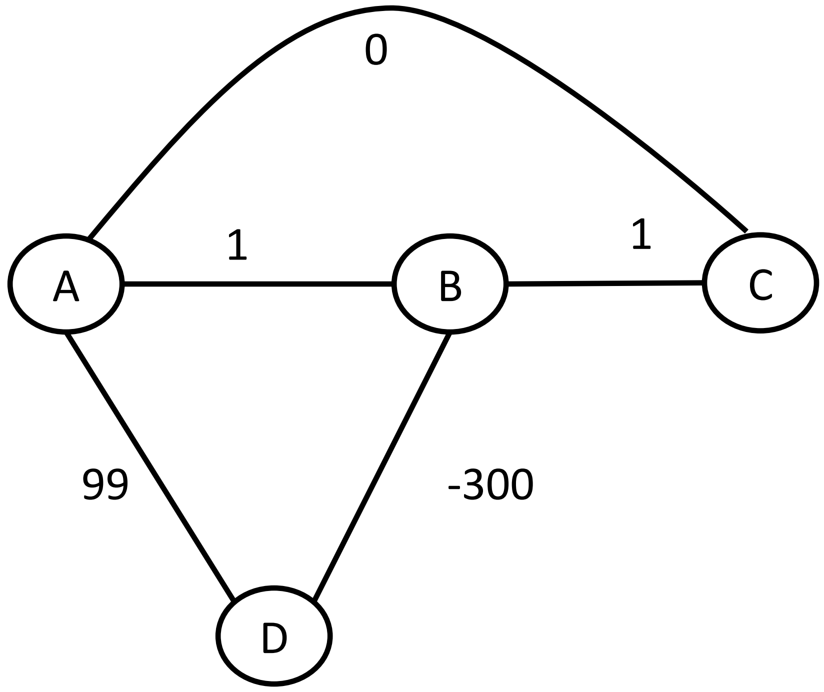
Algorithm Dijkstra_min_priority_queue

- Procedure Relax decreases the keys in the min-priority queue Q since it may change an element of array `distance`
- If we implement the min-priority queue `Q` simply by using the array `distance` and each time we look for the minimum value in the array distance, then `Extract-Min` takes $O(|V|)$ time, so overall Dijkstra's algorithm requires $O(|E| + |V| \cdot |V|) = O(|V|^2)$ time
- If we implement the min-priority queue Q using a Fibonacci heap, Dijkstra's algorithm requires $O(|E| + |V| \cdot \log|V|)$ time

- Dijkstra’s algorithm (either Dijkstra_least_weight or Dijkstra_min_priority_queue) does not work for graphs that have cycles of negative weight
- Example:



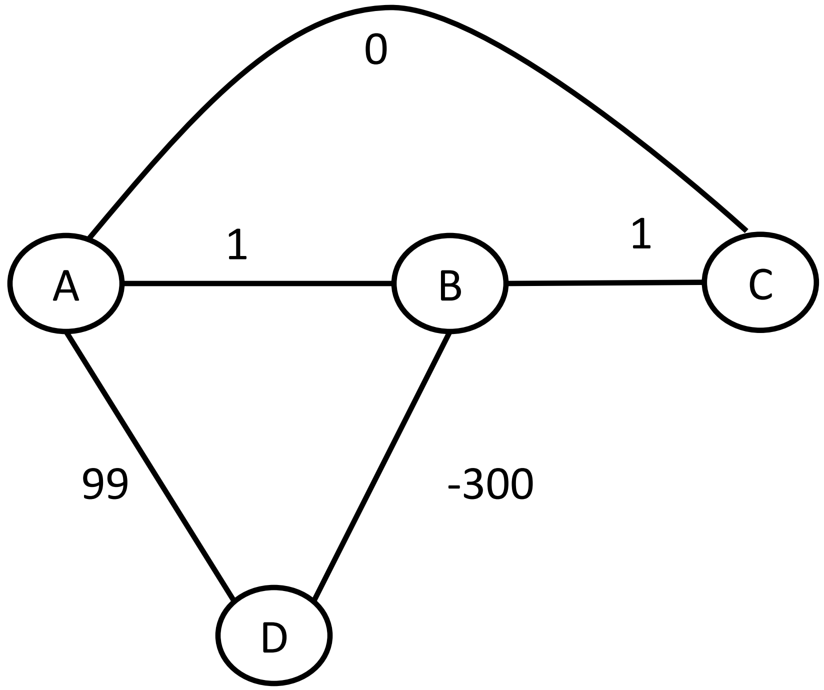
- Dijkstra_least_weight does not work for graphs that have cycles of negative weight



	A	B	C	D	
distance	0	1	0	-299	incorrect
distance	0	-201	-200	-299	correct

- First, you set distances to ∞, seen [ ] to all false, and penultimate [ ] to all nil
- Select node A: distance[A] = 0, penultimate[A]=A, seen[A] = true
- Select the edge {A,C} and set distance[C]= 0, penultimate[C]=A, and seen[C]=true.
- Select the edge {A,B} and set distance[B]=1, penultimate[B]=A, seen[B]=true
- Select edge {B,D} and set distance[D]=-299, penultimate[D]=B, seen[D]=true

- Dijkstra_min_priority_queue does not work for graphs that have cycles of negative weight

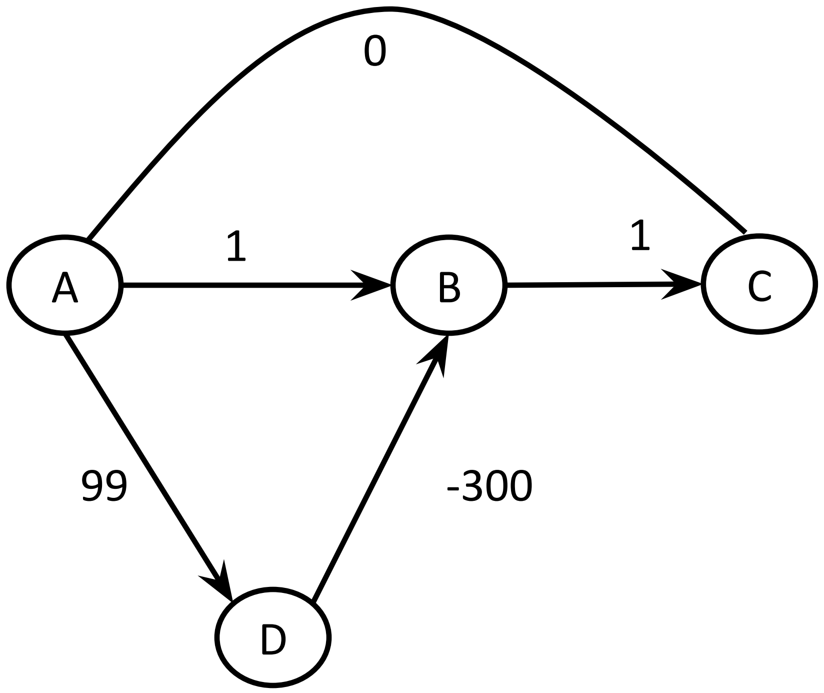


- First, set distance[A] = 0 and the other distances to ∞, array `penultimate` to all nil
- Then set array `seen` to all false
- Extract node A, set seen[A]=true and relax all edges incident to A: distance[B]= 1, distance[C]=0, distance[D]=99, array `penultimate` to all A
- Extract node C, set seen[C]=true and relax all edges incident to A: no changes
- Extract node B, set seen[B]=true and relax all edges incident to B: distance[D]=-299, penultimate[D]=B
- Extract node D, set seen[D]=true and relax all edges incident to D: no changes

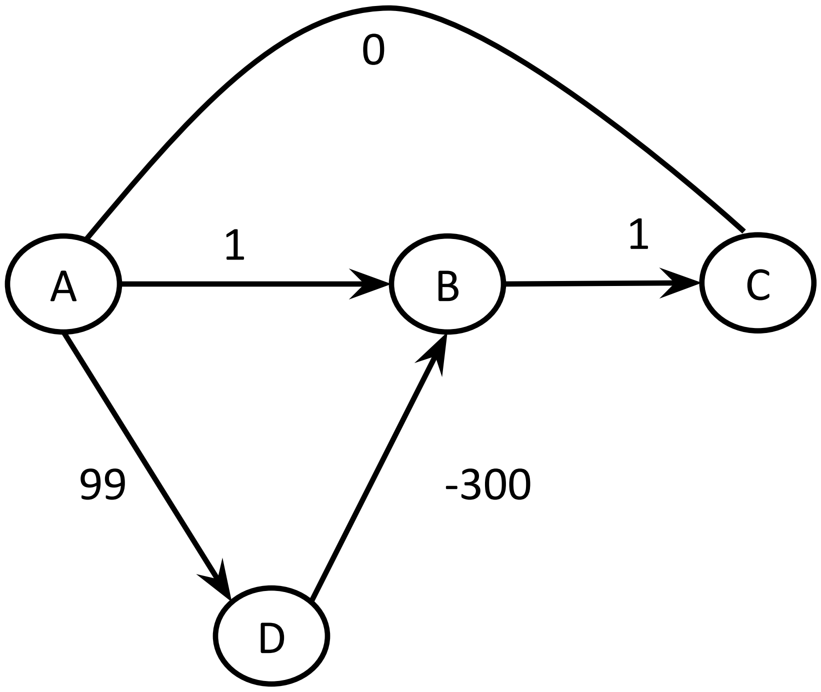
	A	B	C	D	
distance	0	1	0	-299	**incorrect**
distance	0	-201	-200	-299	**correct**

- Notice that at the end of Dijkstra's algorithm , distance[B] is still 1, even though the shortest path to B has length -201, and distance[C] is still 0, even though the shortest path to C has length -200, and Dijkstra’s algorithm thus fails to accurately compute distances for negative weights.
- Dijkstra’s algorithm does not work for directed graphs that have negative weight edges

- Dijkstra’s algorithm does not work for directed graphs that have negative weight edges
- Example:



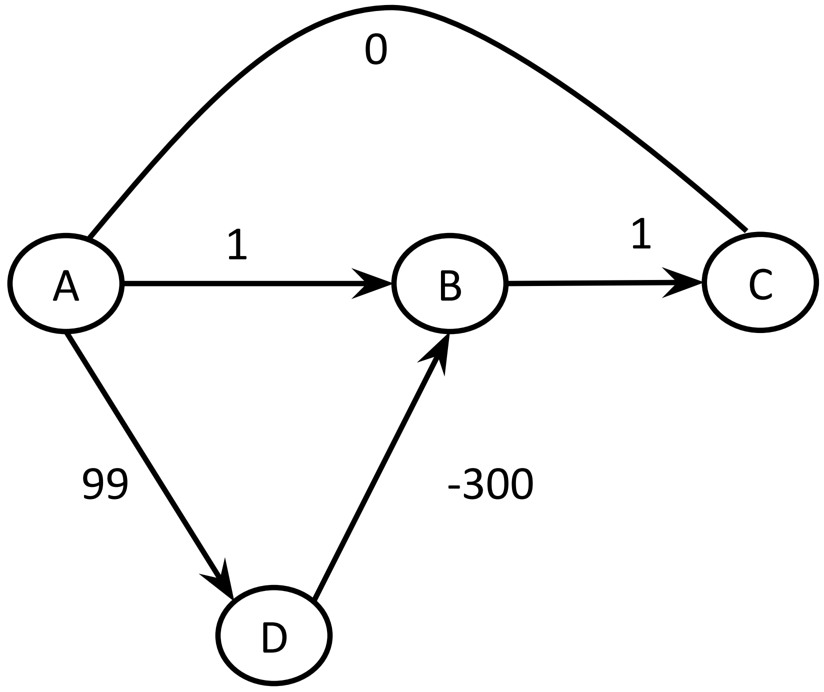
- Dijkstra_least_weight does not work for directed graphs with negative weight edges



	A	B	C	D	
distance	0	1	0	-299	incorrect
distance	0	-201	-200	-299	correct

- First, you set distances to ∞, seen [ ] to all false, and penultimate [ ] to all nil
- Select node A: distance[A] = 0, penultimate[A]=A, seen[A] = true
- Select the edge {A,C} and set distance[C]= 0, penultimate[C]=A, and seen[C]=true.
- Select the edge {A,B} and set distance[B]=1, penultimate[B]=A, seen[B]=true
- Select edge {B,D} and set distance[D]=-299, penultimate[D]=B, seen[D]=true

- Dijkstra_min_priority_queue does not work for directed graphs with negative weight edges



- First, set distance[A] = 0 and the other distances to ∞, array `penultimate` to all nil
- Then set array `seen` to all false
- Extract node A, set seen[A]=true and relax all edges incident to A: distance[B]= 1, distance[C]=0, distance[D]=99, array `penultimate` to all A
- Extract node C, set seen[C]=true and relax all edges incident to A: no changes
- Extract node B, set seen[B]=true and relax all edges incident to B: distance[D]=-299, penultimate[D]=B
- Extract node D, set seen[D]=true and relax all edges incident to D: no changes

	A	B	C	D	
distance	0	1	0	-299	**incorrect**
distance	0	-201	-200	-299	**correct**

- Notice that at the end of Dijkstra's algorithm , distance[B] is still 1, even though the shortest path to B has length -201, and distance[C] is still 0, even though the shortest path to C has length -200, and Dijkstra’s algorithm thus fails to accurately compute distances for negative weights.
- Example of a directed graph with negative and positive weights: the energy spent/gain when hiking various trails in a mountain
- When hiking uphill, you lose energy so the uphill arc will have a negative value for the energy
- When hiking downhill, you gain energy so the downhill arc will have a positive value for the energy

- The best-known algorithm for finding single-source shortest paths in a directed graph with negative edge weights is the *Bellman-Ford algorithm* but it requires that the graph has no negative value cycles.
- However, Bellman-Ford algorithm requires $O(|V| \cdot |E|)$ time, while Dijkstra's algorithm requires $O(|E| + |V| \cdot \log|V|)$ time, which is asymptotically faster for graphs where E is $O(|V|)$ and dense graphs (where E is $O(|V|^2)$

Bellman-Ford-Moore (BFM) Algorithm

- If a graph contains a negative weight cycle, then there is no shortest paths between any two points lying on the cycle: any path can be made cheaper by one more run around the negative cycle
- BFM algorithm can detect negative cycles
- BFM is based on the principle of relaxation, like Dijkstra’s algorithm: an approximation to the correct distance between two points is gradually replaced by smaller values until eventually reaching the smallest value (an optimum solution)
- BFM algorithm relaxes all the edges, and does this |V|-1 times, where |V| is the number of vertices

- BFM algorithm:

```
Initialize(distance, penultimate, s)
  for i=1 to |V|-1 do {
      for each vertex v adjacent to u in E do
          Relax (u, v, distance, penultimate)
}

Procedure Relax(x,y, distance, penultimate)
  if ( distance[y] = ∞ or distance[y] > distance [x] + weight({x,y}) ) then
     distance [y] = distance [x] + weight({x,y})
     penultimate [y] = x

Procedure Initialize(distance, penultimate, s)
  distance[ ] = ∞
  penultimate[ ] = nil (or some other value)
  distance[s] = 0
```

- Since the longest possible path without a cycle has no more than |V|-1 edges, the edges must be scanned |V|-1 times to ensure the shortest path has been found for all nodes.
- Bellman-Ford-Moore's algorithm uses dynamic programming to explore the graph.
- Dijkstra's algorithm uses the greedy approach to select the closest unvisited node to be explored next.