# Dictionary ADT

Key-value pair (KVP)

- a *key*
- some *data* (the "value")

and is called a *key-value pair* (KVP). Keys can be compared and are (typically) unique.

Operations:

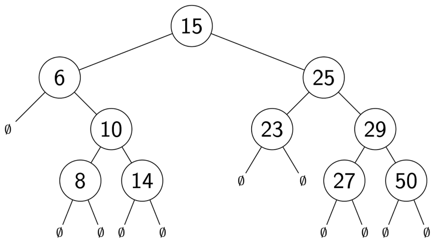
- *search*($k$) (also called *findElement*($k$))
- *insert*($k, v$) (also called *insertItem*($k, v$))
- *delete*($k$) (also called *removeElement*($k$)))
- optional: *closestKeyBefore*, *join*, *isEmpty*, *size*, *etc.*

- **Unordered array or linked list**: $\Theta(1)$ insert, $\Theta(n)$ search and delete
- **Ordered array**: $\Theta(\log n)$ search, $\Theta(n)$ insert and delete
- **Binary search trees**: $\Theta(height)$ search, insert and delete
- **Balanced BST** (AVL trees): $\Theta(\log n)$ search, insert, and delete

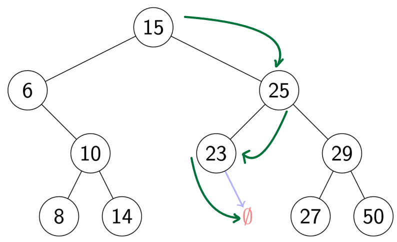
# Binary Search Tree (Review)

Structure: Binary tree: all nodes have two (possibly empty) subtrees
Every node stores a KVP
Empty subtrees usually not shown

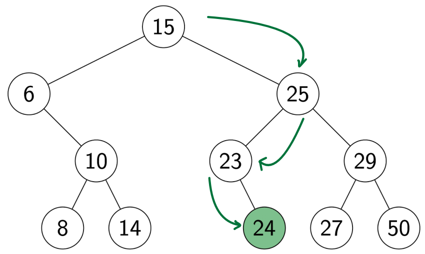
Ordering: Every key $k$ in $T.left$ is less than the root key.
Every key $k$ in $T.right$ is greater than the root key.



*BST::search*($k$) Start at root, compare $k$ to current node's key.
Stop if found or subtree is empty, else recurse at subtree.

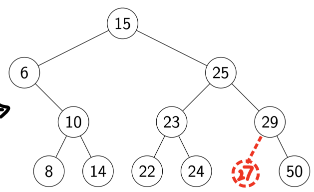


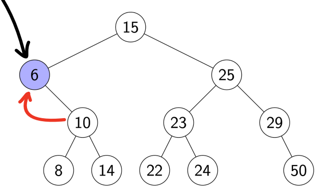
*BST::insert*($k, v$) Search for $k$, then insert $(k, v)$ as new node

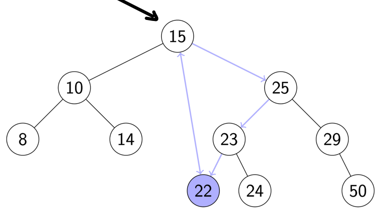


# Delete in BST

- First search for the node $x$ that contains the key.
- If $x$ is a **leaf** (both subtrees are empty), delete it.
- If $x$ has one non-empty subtree, move child up
- Else, swap key at $x$ with key at **successor** or **predecessor** node and then delete that node







## Height of BST:

Worst case: $n-1 = \Theta(n)$

Best case: $\Theta(\log n)$

Average case: $\Theta(\log n)$

# AVL Tree: BST+ height-Balance property

Introduced by Adel'son-Vel'skiĭ and Landis in 1962, an **AVL Tree** is a BST with an additional **height-balance property**:

*The heights of the left and right subtree differ by at most 1.*

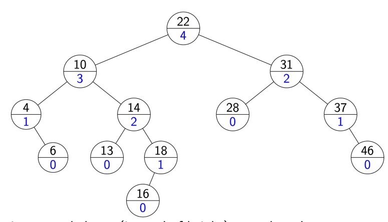
(The height of an empty tree is defined to be $-1$.)

If node $v$ has left subtree $L$ and right subtree $R$, then

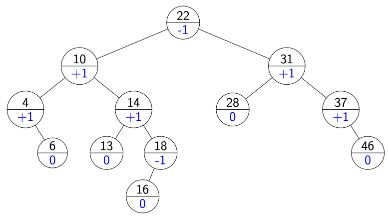
$$\textbf{balance}(v) := height(R) - height(L) \in \{-1, 0, 1\}:$$

- $-1$ means $v$ is *left-heavy*
- $0$ means $v$ is *balanced*
- $+1$ means $v$ is *right-heavy*

(The lower numbers indicate the height of the subtree.)



Alternative: store balance (instead of height) at each node.



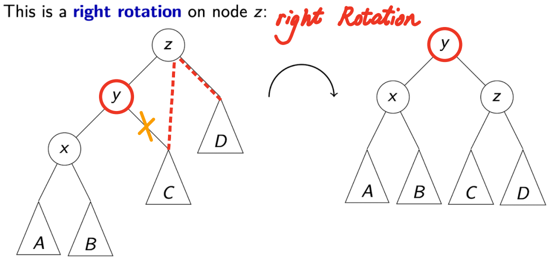
- Need to store at each node $v$ the height of the subtree rooted at it
- Can show: It suffices to store $balance(v)$ instead
  - uses fewer bits, but code gets more complicated

**Theorem:** An AVL tree on $n$ nodes has $\Theta(\log n)$ height.
$\Rightarrow$ *search*, *insert*, *delete* all cost $\Theta(\log n)$ in the *worst case!*

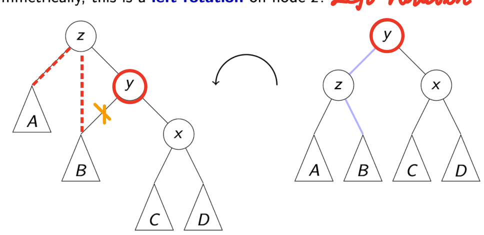
Height of AVL Tree is $\Theta(\log n)$
worst case time for search, insert, delete is $\Theta(\log n)$

## AVL Rotation:

This is a **right rotation** on node $z$: right Rotation



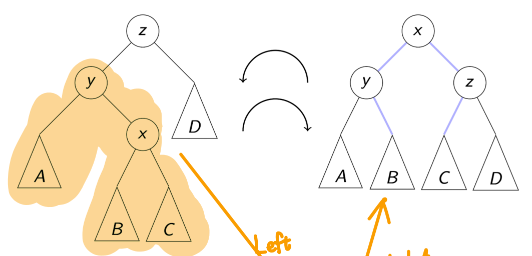
Symmetrically, this is a **left rotation** on node $z$: Left Rotation



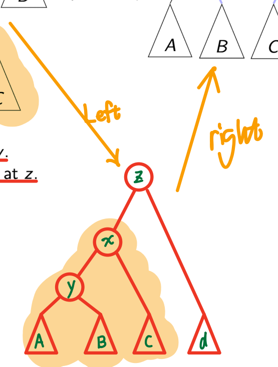
```
rotate-right(z)
    y ← z.left, z.left ← y.right, y.right ← z
    setHeightFromSubtrees(z), setHeightFromSubtrees(y)
    return y // returns new root of subtree
```

### Double right Rotation

This is a **double right rotation** on node $z$:

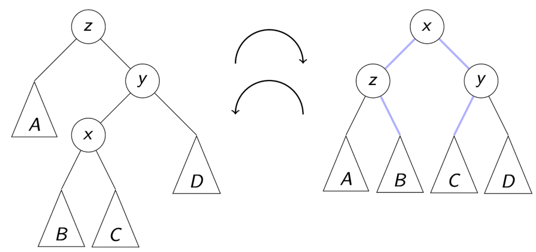


First, a left rotation at $y$.
Second, a right rotation at $z$.




### Double left rotation

Symmetrically, there is a **double left rotation** on node $z$:



First, a right rotation at $y$.
Second, a left rotation at $z$.

```
restructure(x, y, z)
node x has parent y and grandparent z
    case
        : // Right rotation
          return rotate-right(z)
        : // Double-right rotation
          z.left ← rotate-left(y)
          return rotate-right(z)
        : // Double-left rotation
          z.right ← rotate-right(y)
          return rotate-left(z)
        : // Left rotation
          return rotate-left(z)
```

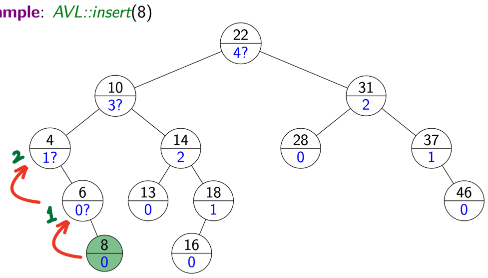
**Rule:** The middle key of $x, y, z$ becomes the new root.

## AVL insertion:

```
AVL::insert(k, v)
    z ← BST::insert(k, v)   // leaf where k is now stored
    while (z is not NIL)
        if (|z.left.height − z.right.height| > 1) then   左比右更深
            Let y be taller child of z
            Let x be taller child of y (break ties to avoid zigzag)
            z ← restructure(x, y, z) // see later
            break          // can argue that we are done
        setHeightFromSubtrees(z)
        z ← z.parent   ←———  一路rotate上去
```

```
setHeightFromSubtrees(u)
    u.height ← 1 + max{u.left.height, u.right.height}
```

**Example:** *AVL::insert*(8)



# AVL Deletion:

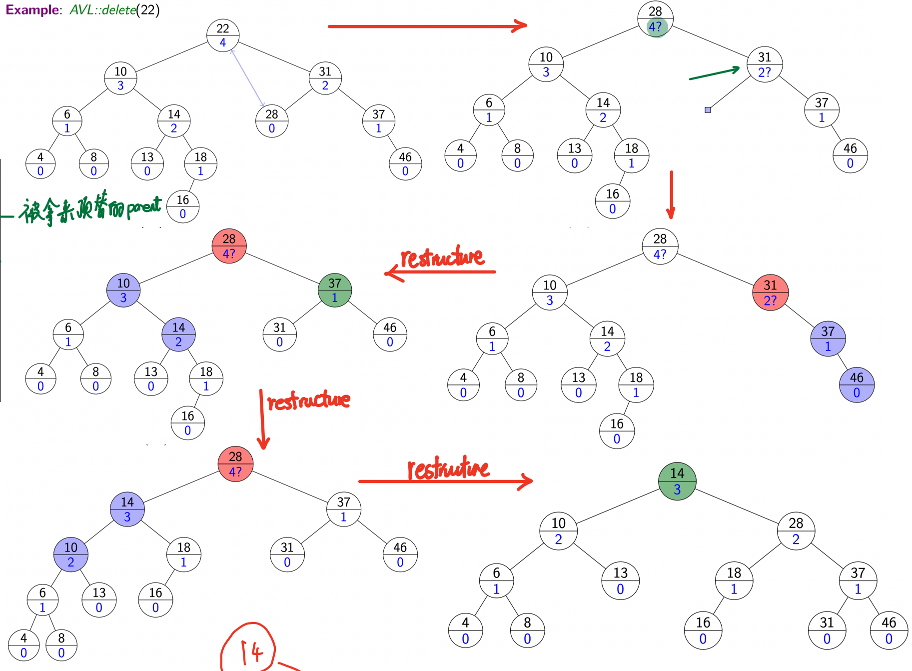
Remove the key $k$ with *BST::delete*.
Find node where *structural* change happened.
(This is not necessarily near the node that had $k$.)
Go back up to root, update heights, and rotate if needed.

Could be ambiguous: y's children could have equal height. In that case, which x should we pick?

*AVL::delete*($k$)
1. $z \leftarrow$ *BST::delete*($k$)
2. // Assume $z$ is the parent of the BST node that was removed ← 被拿去顶替的parent
3. **while** ($z$ is not NIL)
4. **if** ($|z.left.height - z.right.height| > 1$) **then** 左比右更深
5. Let $y$ be taller child of $z$
6. Let $x$ be taller child of $y$ (break ties to avoid zig-zag)
7. $z \leftarrow$ *restructure*($x, y, z$)
8. // *Always* continue up the path and fix if needed.
9. *setHeightFromSubtrees*($z$)
10. $z \leftarrow z.parent$

If the balance factor of y equals 0, then: single rotation vs. double rotation. Single rotations are preferred because they require fewer steps.
Thus if possible, choose x where: x > y > z or x < y < z, because these leads to single rotations.
ONLY IF the balance factor of y has opposite sign to that of z, should a double rotation be preferred.

**Example**: *AVL::delete*(22)




## AVL operation cost:

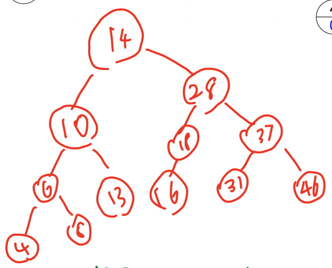
AVL search: $\Theta(\text{height}) = \Theta(\log n)$

AVL insert: $\Theta(\text{height}) = \Theta(\log n)$

AVL-fix will be called at most once

AVL delete: $\Theta(\text{height}) = \Theta(\log n)$

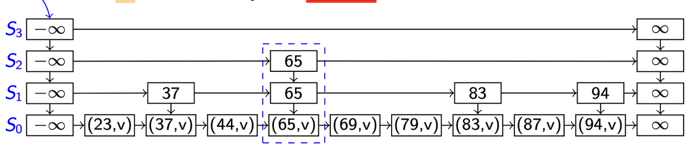
AVL-fix will be called at most $\Theta(\text{height})$ times



*Worst-case* cost for all operations is $\Theta(height) = \Theta(\log n)$.

# Skip List:

- A hierarchy $S$ of ordered linked lists (*levels*) $S_0, S_1, \cdots, S_h$:
  - Each list $S_i$ contains the special keys $-\infty$ and $+\infty$ (sentinels)
  - List $S_0$ contains the KVPs of $S$ in non-decreasing order.
    (The other lists store only keys, or links to nodes in $S_0$.)
  - Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_h$
  - List $S_h$ contains only the sentinels



- Each KVP belongs to a **tower** of nodes
- There are (usually) more *nodes* than *keys*
- The skip list consists of a reference to the topmost left node.
- Each node $p$ has references $p.after$ and $p.below$

* 每层必有 $-\infty$ 和 $\infty$
* 最顶层只有 $-\infty$ 和 $\infty$
* None-decreasing order. (不会更小)
* 下层包含上层.

* Expect space: $O(n)$
* Expect height: $O(\log n)$

height at most $3\log n$, chance $\geq 1-\frac{1}{n^2}$

*skipList::search*: $O(\log n)$ expected time
- # drop-downs = height
- expected # scan-forwards is $\leq 2$ in each level

*skipList::insert*: $O(\log n)$ expected time

*skipList::delete*: $O(\log n)$ expected time

## Skip List: Get predecessor

* (node before where k should be)

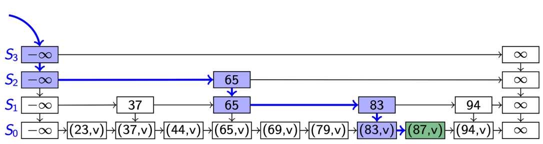
*getPredecessors* ($k$)
1. $p \leftarrow$ topmost left sentinel
2. $P \leftarrow$ stack of nodes, initially containing $p$
3. **while** $p.below \neq$ NIL **do**
4. $p \leftarrow p.below$
5. **while** $p.after.key < k$ **do** $p \leftarrow p.after$
6. $P.push(p)$
7. **return** $P$

如果下面不是null就往下走,
如果后面的比k小就往后走

## Skip List: Search

*skipList::search* ($k$)
1. $P \leftarrow$ *getPredecessors*($k$)
2. $p_0 \leftarrow P.top()$ // predecessor of $k$ in $S_0$
3. **if** $p_0.after.key = k$ **return** $p_0.after$
4. **else return** "not found, but would be after $p_0$"

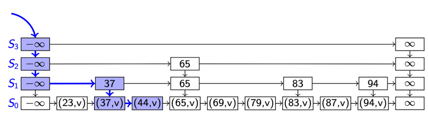
**Example**: *search*(87)

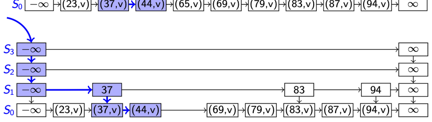


## Skip List: Delete

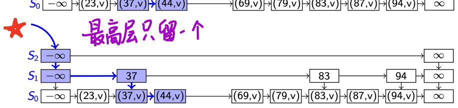
*skipList::delete*($k$)
1. $P \leftarrow$ *getPredecessors*($k$)
2. **while** $P$ is non-empty
3. $p \leftarrow P.pop()$ // predecessor of $k$ in some layer
4. **if** $p.after.key = k$
5. $p.after \leftarrow p.after.after$
6. **else break** // no more copies of $k$
7. $p \leftarrow$ topmost left sentinel
8. **while** $p.below.after$ is the $\infty$-sentinel
   // the two top lists are both only sentinels, remove one
9. $p.below \leftarrow p.below.below$
10. $p.after.below \leftarrow p.after.below.below$

Example: *skipList::delete*(65)
*getPredecessors*(65)





最高层只留一个



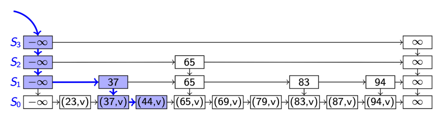
## Skip List: Insert

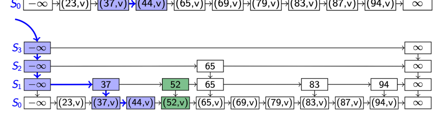
*skipList::insert*($k, v$)
- Randomly repeatedly toss a coin until you get tails
- Let $i$ the number of times the coin came up heads; this will be the height of the tower of $k$

$$P(\text{tower of key } k \text{ has height} \geq i) = \left(\frac{1}{2}\right)^i$$

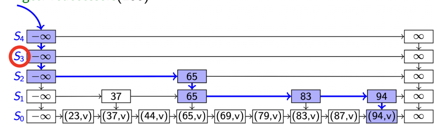
- Increase height of skip list, if needed, to have $h > i$ levels.
- Use *getPredecessors*($k$) to get stack $P$.
  The top $i$ items of $P$ are the predecessors $p_0, p_1, \cdots, p_i$ of where $k$ should be in each list $S_0, S_1, \cdots, S_i$
- Insert $(k, v)$ after $p_0$ in $S_0$, and $k$ after $p_j$ in $S_j$ for $1 \leq j \leq i$

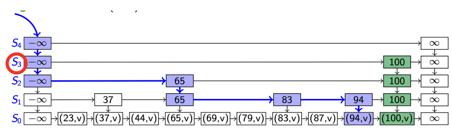
Example: *skipList::insert*(52, $v$)
Coin tosses: H,T $\Rightarrow i = 1$
*getPredecessors*(52)





Example: *skipList::insert*(100, $v$)
Coin tosses: H,H,H,T $\Rightarrow i = 3$
*Height increase*
*getPredecessors*(100)





# Module 5

**Theorem**: In the comparison model (on the keys), $\Omega(\log n)$ comparisons are required to search a size-$n$ dictionary.

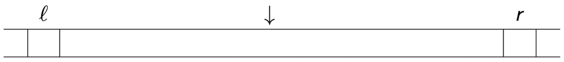
## Binary Search:

Recall the run-times in a *sorted array*:
- *insert, delete*: $\Theta(n)$
- *search*: $\Theta(\log n)$

```
Binary-search(A, n, k)
A: Sorted array of size n, k: key
    ℓ ← 0
    r ← n − 1
    while (ℓ < r)
        m ← ⌊(ℓ+r)/2⌋
        if (A[m] < k) then ℓ = m + 1
        else if (k < A[m]) then r = m − 1
        else return m
    if (k = A[ℓ]) return ℓ
    else return "not found, but would be between ℓ−1 and ℓ"
```

## Interpolation Search:

*binary-search*$(A[\ell, r], k)$: Compare at index $\lfloor \frac{\ell+r}{2} \rfloor = \ell + \lfloor \frac{1}{2}(r-\ell) \rfloor$ 取中点




*interpolation-search*$(A[\ell, r], k)$: Compare at index $\ell + \left\lfloor \frac{k - A[\ell]}{A[r]-A[\ell]}(r-\ell) \right\rfloor$ 取"k"应该在的位置


- Code very similar to binary search, but compare at interpolated index
- Need a few extra tests to avoid crash due to $A[\ell] = A[r]$

```
interpolation-search(A, n, k)
A: Sorted array of size n, k: key
    ℓ ← 0
    r ← n − 1
    while (ℓ < r)&&(A[r]! = A[ℓ])&&(k ≥ A[ℓ])&&(k ≤ A[r]))
        m ← ℓ + ⌊(k−A[ℓ])/(A[r]−A[ℓ]) · (r − ℓ)⌋
        if (A[m] < k) then ℓ = m + 1
        else if (k < A[m]) then r = m − 1
        else return m
    if (k = A[ℓ]) return ℓ
    else return "not found, but would be between ℓ−1 and ℓ"
```

Works well if keys are uniformly distributed.
- Can show: the array in which we ~~recurse into has size $\sqrt{n}$ on average.~~
- Recurrence relation is $T^{(avg)}(n) = T^{(avg)}(\sqrt{n}) + \Theta(1)$.
- This resolves to $T^{(avg)}(n) \in \Theta(\log\log n)$.

Worse Case: $\Theta(n)$

## Trie / radix tree

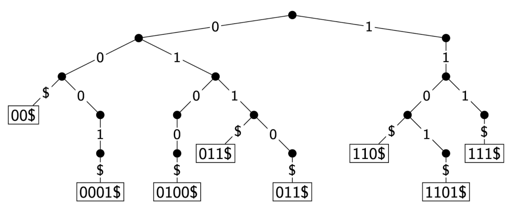
Dictionary for bitstrings

- A tree based on *bitwise comparisons*: Edge labelled with corresponding bit
- Similar to *radix sort*: use individual bits, not the whole key

**Assumption:** Dictionary is **prefix-free**: no string is a prefix of another

(A **prefix** 前缀 of a string $S[0..n-1]$ is a substring $S[0..i-1]$ for some $0 \le i \le n$.)
- Assumption satisfied if all strings have the same length.
- Assumption satisfied if all strings end with 'end-of-word' character \$.

有一个就行




Then items (keys) are stored *only* in the leaf nodes

## Trie: Search

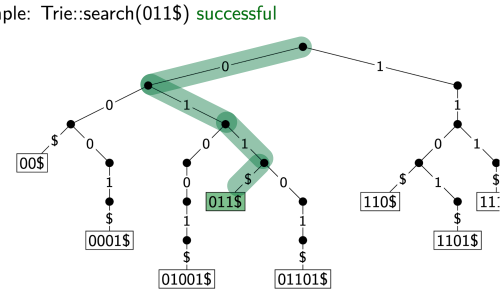
- start from the root and the most significant bit of $x$
- follow the link that corresponds to the current bit in $x$; return failure if the link is missing
- return success if we reach a leaf (it must store $x$)
- else recurse on the new node and the next bit of $x$

```
Trie::search(v ← root, d ← 0, x)
v: node of trie; d: level of v, x: word stored as array of chars
    if v is a leaf
        return v
    else
        let c be child of v labelled with x[d]
        if there is no such child
            return "not found"
        else Trie::search(c, d + 1, x)
```

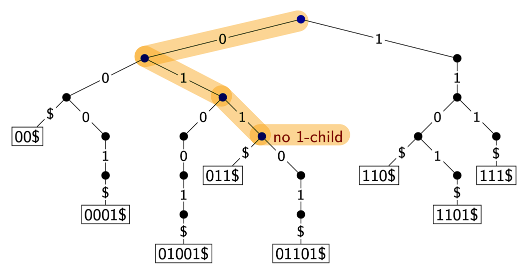
## Time Complexity for all Operation: $\Theta(|x|)$

$|x|$ = length of string

Example: Trie::search(011\$) successful



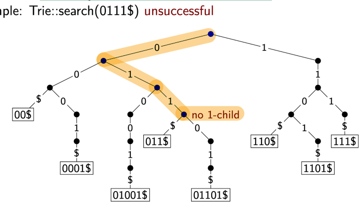

Example: Trie::search(0111\$) unsuccessful



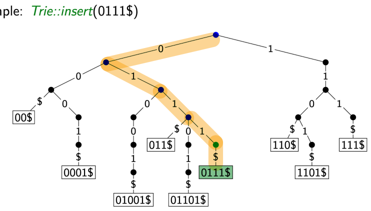

## Trie: Search

- *Trie::insert(x)*
  - Search for $x$, this should be unsuccessful
  - Suppose we finish at a node $v$ that is missing a suitable child. Note: $x$ has extra bits left.
  - Expand the trie from the node $v$ by adding necessary nodes that correspond to extra bits of $x$.

Example: Trie::search(0111\$) unsuccessful



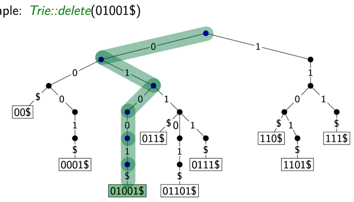

Example: *Trie::insert*(0111\$)

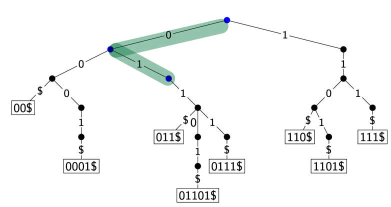


## Trie: Delete

- *Trie::delete*$(x)$
  - Search for $x$
  - let $v$ be the leaf where $x$ is found
  - delete $v$ and all ancestors of $v$ until we reach an ancestor that has two children.

Example: *Trie::delete*(01001\$)

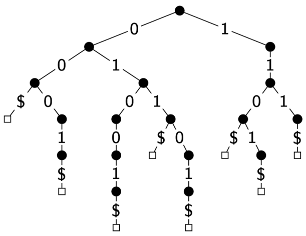




## Trie Version 1: No leaf labels

Do not store actual keys at the leaves.

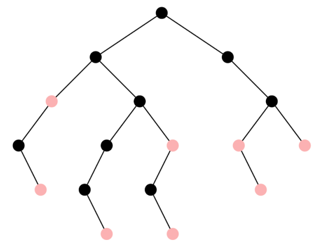
- The key is stored implicitly through the characters along the path to the leaf. It therefore need not be stored again.
- This halves the amount of space needed.




## Trie Version 2: Allow proper Prefixes

Allow prefixes to be in dictionary.

- Internal nodes may now also represent keys.
  Use a *flag* to indicate such nodes.
- No need for end-of-word character \$
- Now a trie of bitstrings is a binary tree. Can express 0-child and 1-child implicitly via left and right child. ?
- More space-efficient.

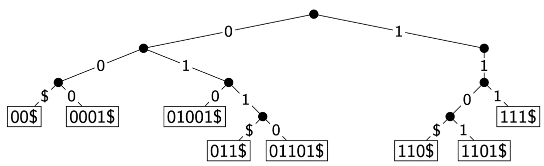


## Trie Version 3: Pruned Trie

**Pruned Trie:** Stop adding nodes to trie as soon as the key is unique.

- A node has a child only if it has at least two descendants.
- Note that now we *must* store the full keys (why?)
- Saves space if there are only few bitstrings that are long.
- Could even store infinite bitstrings (e.g. real numbers)

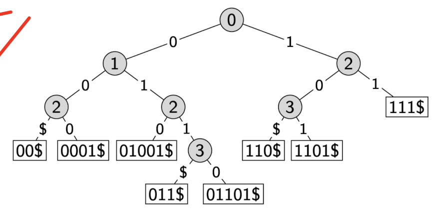
Most efficient




## Trie Version 4: Compressed Trie (Patricia-Tries)

**Compressed Trie:** compress paths of nodes with only one child

- Each node stores an *index*, corresponding to the depth in the uncompressed trie.
  - This gives the next bit to be tested during a search
- A compressed trie with $n$ keys has at most $n-1$ internal nodes




## Compressed Trie: Search

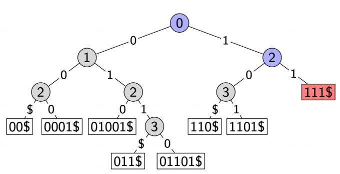
- start from the root and the bit indicated at that node
- follow the link that corresponds to the current bit in $x$; return failure if the link is missing
- if we reach a leaf, expicitly check whether word stored at leaf is $x$
- else recurse on the new node and the next bit of $x$

Time Complexity for all Operation: $\theta(|x|)$

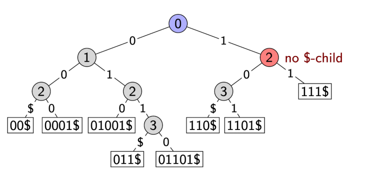
$|x|$ = length of string

```
CompressedTrie::search(v ← root, x)
v: node of trie; x: word
1.  if v is a leaf
2.      return strcmp(x, v.key)
3.  else
4.      d ← index stored at v
5.      c ← child of v labelled with x[d]
6.      if there is no such child
7.          return "not found"
8.      else CompressedTrie::search(c, x)
```

Example: CompressedTrie::search(101\$) unsuccessful




Example: CompressedTrie::search(10\$) unsuccessful




## Compressed Trie: Delete (x):

- Perform *search*($x$)
- Remove the node $v$ that stored $x$
- Compress along path to $v$ whenever possible.

## Compressed Trie: insert (x)

- Perform *search*($x$)
- Let $v$ be the node where the search ended.
- Conceptually simplest approach:
  - Uncompress path from root to $v$.
  - Insert $x$ as in an uncompressed trie.
  - Compress paths from root to $v$ and from root to $x$.

  But it can also be done by only adding those nodes that are needed, see the textbook for details.

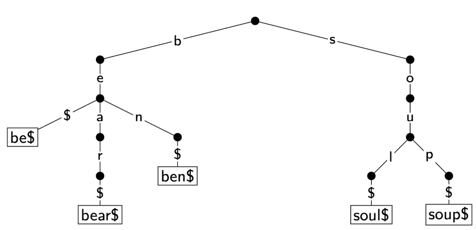
**Solution 1:** Array of size $|\Sigma|+1$ for each node.
Complexity: $O(1)$ time to find child, $O(|\Sigma|n)$ space.

**Solution 2:** List of children for each node.
Complexity: $O(|\Sigma|)$ time to find child, $O(\#\text{children})$ space.

**Solution 3:** Dictionary (AVL-tree?) of children for each node.
Complexity: $O(\log(\#\text{children}))$ time, $O(\#\text{children})$ space.

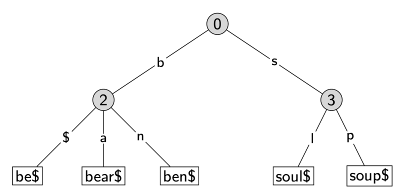
## Multiway Trie:

- To represent *strings* over any *fixed alphabet* $\Sigma$
- Any node will have at most $|\Sigma|+1$ children (one child for the end-of-word character \$)
- Example: A trie holding strings {bear\$, ben\$, be\$, soul\$, soup\$}




## Compressed Multiway Trie:

- **Variation**: Compressed multi-way tries: compress paths as before
- Example: A compressed trie holding strings {bear\$, ben\$, be\$, soul\$, soup\$}



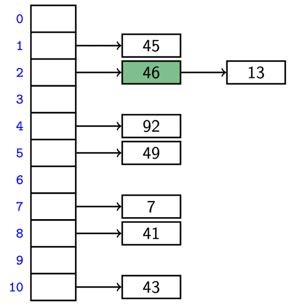

- Operations *search*($x$), *insert*($x$) and *delete*($x$) are exactly as for tries for bitstrings.

Time Complexity for all Operation: $\theta(|x| \cdot \text{time to find child})$

$|x|$ = length of string

# Hashing

$M = 11$, $h(k) = k \bmod 11$




*insert*(46)

$h(46) = 2$

## Seperate chaining:

- *search*$(k)$: Look for key $k$ in the list at $T[h(k)]$. Apply MTF-heuristic!
- *insert*$(k, v)$: Add $(k, v)$ to the front of the list at $T[h(k)]$.
- *delete*$(k)$: Perform a search, then delete from the linked list.

| | Seperate chaining | |
|---|---|---|
| insert | $O(1)$ | |
| Search | $\Theta$(1+ Size of Bucket) | |
| delete | $\Theta$(1+ Size of Bucket) | |

$M$ = Table size

$n$ = # of item in total

- The *average* bucket-size is $\frac{n}{M} =: \alpha$. ($\alpha$ is also called the **load factor**.)

Load factor: $\alpha = \frac{n}{M} = \frac{\text{\# of item in total}}{\text{Table size}}$

## Rehashing:

* Rehash when $\alpha$ getting too big or too small

- For all collision resolution strategies, the run-time evaluation is done in terms of the *load factor* $\alpha = n/M$.
- We keep the load factor small by **rehashing** when needed:
  - Keep track of $n$ and $M$ throughout operations
  - If $\alpha$ gets too large, create new (twice as big) hash-table, new hash-function(s) and re-insert all items in the new table.
- Rehashing costs $\Theta(M + n)$ but happens rarely enough that we can ignore this term when amortizing over all operations.
- We should also re-hash when $\alpha$ gets too small, so that $M \in \Theta(n)$ throughout, and the space is always $\Theta(n)$.

**Summary:** If we maintain $\alpha \in \Theta(1)$, then (under the uniform hashing assumption) the average cost for hashing with chaining is $O(1)$ and the space is $\Theta(n)$.

## Open addressing: 不允许一个 spot 多个元素, 但允许一个 key 出现在多个 slot.

### Linear Probing

$h(k, i) = (h(k) + i) \bmod M$, for some hash function $h$.

如果一个 spot 有东西, 放到下一个去.

Idea 1: Move later items in the probe sequence forward.

delete: 一般 Idea 2: **lazy deletion**: Mark spot as *deleted* (rather than `NIL`) and continue searching past deleted spots.

## Cuckoo Hashing:

We use two independent hash functions $h_0, h_1$ and two tables $T_0, T_1$.

**Main idea:** An item with key $k$ can *only* be at $T_0[h_0(k)]$ or $T_1[h_1(k)]$.

- *search* and *delete* then take constant time.
- *insert always* initially puts a new item into $T_0[h_0(k)]$

  If $T_0[h_0(k)]$ is occupied: "kick out" the other item, which we then attempt to re-insert into its alternate position $T_1[h_1(k)]$

  This may lead to a loop of "kicking out". We detect this by aborting after too many attempts.

  In case of failure: rehash with a larger $M$ and new hash functions.

*insert* may be slow, but is expected to be constant time if the load factor is small enough.

- The two hash-tables need not be of the same size.
- *Load factor* $\alpha = n/(\text{size of } T_0 + \text{size of } T_1)$
- One can argue: If the load factor $\alpha$ is small enough then insertion has $O(1)$ expected run-time.
- This crucially requires $\alpha < \frac{1}{2}$.

$\alpha = \frac{n}{|T_0| + |T_1|}$

$O(1)$ insertion need $\alpha < \frac{1}{2}$

## Independent Hashing Function:

- Some hashing methods require *two* hash functions $h_0, h_1$.
- These hash functions should be *independent* in the sense that the random variables $P(h_0(k) = i)$ and $P(h_1(k) = j)$ are independent.
- Using two modular hash-functions may often lead to dependencies.
- Better idea: Use *multiplicative method* for second hash function: $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$,
  - $A$ is some floating-point number
  - $kA - \lfloor kA \rfloor$ computes fractional part of $kA$, which is in $[0, 1)$
  - Multiply with $M$ to get floating-point number in $[0, M)$
  - Round down to get integer in $\{0, \ldots, M-1\}$

  Knuth suggests $A = \varphi = \frac{\sqrt{5}-1}{2} \approx 0.618$.

## Double Hashing

- Assume we have two hash independent functions $h_0, h_1$.
- Assume further that $h_1(k) \neq 0$ and that $h_1(k)$ is relative prime with the table-size $M$ for all keys $k$.
  - Choose $M$ prime.
  - Modify standard hash-functions to ensure $h_1(k) \neq 0$

    E.g. modified multiplication method: $h(k) = 1 + \lfloor (M-1)(kA - \lfloor kA \rfloor) \rfloor$
- **Double hashing**: open addressing with probe sequence

$$h(k, i) = h_0(k) + i \cdot h_1(k) \bmod M$$

- *search*, *insert*, *delete* work just like for linear probing, but with this different probe sequence.

| | Seperate chaining | Cuckoo hashing |
|---|---|---|
| insert | $O(1)$ | may be slow / Expect $O(1)$ if $\alpha < \frac{1}{2}$ |
| Search | $\Theta(1 + \text{Size of Bucket})$ | $O(1)$ |
| delete | $\Theta(1 + \text{Size of Bucket})$ | $O(1)$ |

For any open addressing scheme, we *must* have $\alpha < 1$ (why?).
Cuckoo hashing requires $\alpha < 1/2$.

| Avg.-case costs: | *search (unsuccessful)* | *insert* | *search (successful)* |
|---|---|---|---|
| Linear Probing | $\frac{1}{(1-\alpha)^2}$ | $\frac{1}{(1-\alpha)^2}$ | $\frac{1}{1-\alpha}$ |
| Double Hashing | $\frac{1}{1-\alpha}$ | $\frac{1}{1-\alpha}$ | $\frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)$ |
| Cuckoo Hashing | 1 (worst-case) | $\frac{\alpha}{(1-2\alpha)^2}$ | 1 (worst-case) |

**Summary:** All operations have $O(1)$ average-case run-time if the hash-function is uniform and $\alpha$ is kept sufficiently small.
But worst-case run-time is (usually) $\Theta(n)$.

# Range Query:

| | Range Query |
|---|---|
| Unsorted list hash table / array | $\Omega(n)$ |
| Sorted array | $O(\log n + s)$ |
| BST | $O(\text{height} + s)$ |

# Hashing vs. BST

## Advantages of Balanced Search Trees

- $O(\log n)$ worst-case operation cost
- Does not require any assumptions, special functions, or known properties of input distribution
- Predictable space usage (exactly $n$ nodes)
- Never need to rebuild the entire structure
- Supports ordered dictionary operations (rank, select etc.)

## Advantages of Hash Tables

- $O(1)$ operations (if hashes well-spread and load factor small)
- We can choose space-time tradeoff via load factor
- Cuckoo hashing achieves $O(1)$ worst-case for search & delete

# Quadtrees

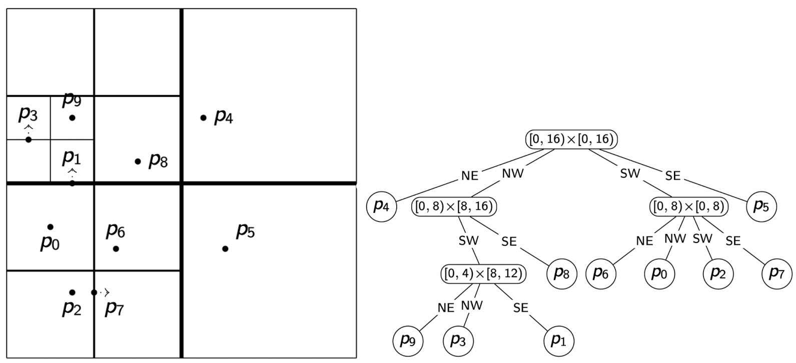
We have $n$ points $S = \{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$ in the plane.

We need a **bounding box** $R$: a square containing all points.

- Can find $R$ by computing minimum and maximum $x$ and $y$ values in $S$
- The width/height of $R$ should be a power of 2

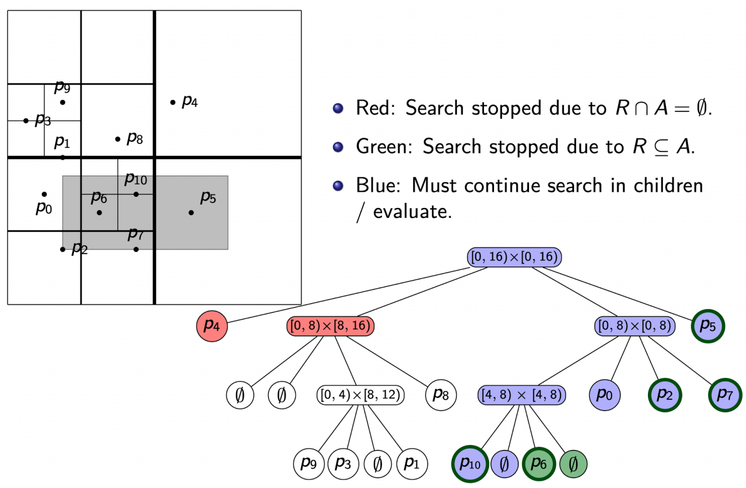
**Structure** (and also how to *build* the quadtree that stores $S$):

- Root $r$ of the quadtree is associated with region $R$
- If $R$ contains 0 or 1 points, then root $r$ is a leaf that stores point.
- Else *split*: Partition $R$ into four equal subsquares (**quadrants**) $R_{NE}, R_{NW}, R_{SW}, R_{SE}$
- Partition $S$ into sets $S_{NE}, S_{NW}, S_{SW}, S_{SE}$ of points in these regions.
  - **Convention:** Points on split lines belong to right/top side




## Quadtree Range Search

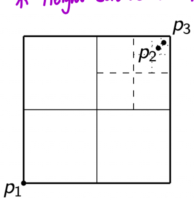
*QTree::RangeSearch*$(r \leftarrow root, A)$
$r$: The root of a quadtree, $A$: Query rectangle
$R \leftarrow$ region associated with node $r$
**if** $(R \subseteq A)$ **then** // inside node
  report all points below $r$; **return**
**if** $(R \cap A$ is empty$)$ **then** // outside node
  **return**
  // The node is a boundary node, recurse
**if** ($r$ is a leaf) **then**
  $p \leftarrow$ point stored at $r$
  **if** $p$ is in $A$ **return** $p$
  **else return**
**for** each child $v$ of $r$ **do**
  *QTree::RangeSearch*$(v, A)$




- Red: Search stopped due to $R \cap A = \emptyset$.
- Green: Search stopped due to $R \subseteq A$.
- Blue: Must continue search in children / evaluate.

Note: We assume here that each node of the quadtree stores the associated square. Alternatively, these could be re-computed during the search (space-time tradeoff).

* Height can be bad.




* Spread Factor of point $S$: $\beta(S) = \frac{\text{sidelength of } R}{\text{minimum distance between points in } S}$

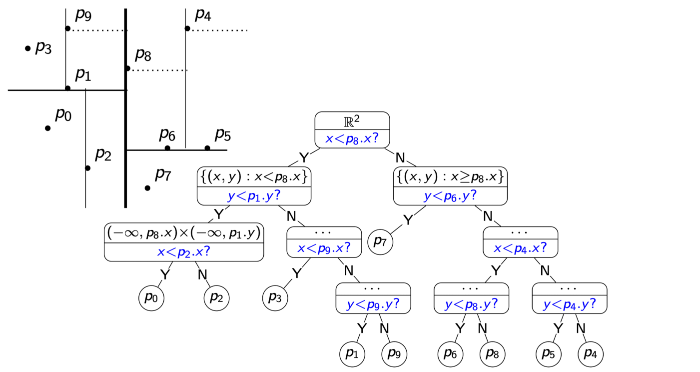
* Height of Quad tree is in $\Theta(\log \beta(S))$

- Complexity to build initial tree: $\Theta(nh)$ worst-case
- Complexity of range search: $\Theta(nh)$ worst-case even if the answer is $\emptyset$

# Kd-tree

- (Point-based) kd-tree idea: Split the region such that (roughly) half the point are in each subtree
- Each node of the kd-tree keeps track of a **splitting line** in one dimension (2D: either vertical or horizontal)
- **Convention:** Points on split lines belong to right/top side
- Continue splitting, switching between vertical and horizontal lines, until every point is in a separate region

(There are alternatives, e.g., split by the dimension that has better aspect ratios for the resulting regions. No details.)




* Height of Kd-tree is $O(\log n)$    * if points share coordinate, height can go $\infty$

| | Quad-tree |
|---|---|
| Build | $\theta(n\log n)$ Time / $O(n)$ space |
| Height | $O(\log n)$ |
| Range search | $O(s+\sqrt{n})$ 2D / $O(s+n^{1-\frac{1}{d}})$ $d$ = dimension |

- *search* (for single point): as in binary search tree using indicated coordinate
- *insert*: search, insert as new leaf.
- *delete*: search, remove leaf and unary parents.

**Problem:** After insert or delete, the split might no longer be at exact median and the height is no longer guaranteed to be $O(\log n)$ even for points in general position.

- **Storage**: $O(n)$
- **Height**: $O(\log n)$
- **Construction time**: $O(n\log n)$
- **Range query time**: $O(s + n^{1-1/d})$

* General position points
* $d$ is constant

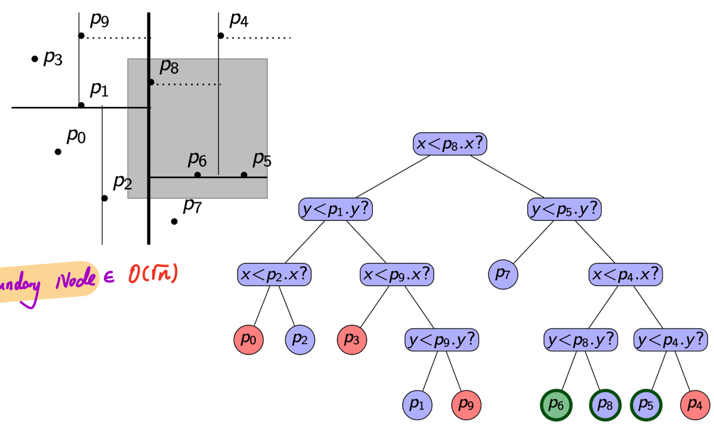
# KDtree: Range Search

- Range search is *exactly* as for quad-trees, except that there are only two children.

*kdTree::RangeSearch*($r \leftarrow root$, $A$)
$r$: The root of a kd-tree, $A$: Query rectangle

```
R ← region associated with node r
if (R ⊆ A) then report all points below r; return
if (R ∩ A is empty) then return
if (r is a leaf) then
    p ← point stored at r
    if p is in A return p
    else return
for each child v of r do
    kdTree::RangeSearch(v, A)
```

* Boundary Node $\in O(\sqrt{n})$

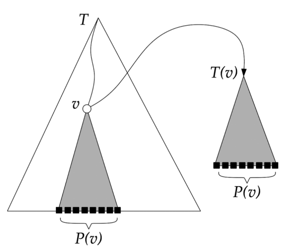



Red: Search stopped due to $R\cap A=\emptyset$. Green: Search stopped due to $R\subseteq A$.

- We assume again that each node stores its associated region.
- To save space, we could instead pass the region as a parameter and compute the region for each child using the splitting line.

# Range Tree

- Somewhat wasteful in space, but much faster range search.
- Have a binary search tree $T$ (sorted by $x$-coordinate); this is the **primary structure**
- Each node $v$ of $T$ has an **associate structure** $T(v)$: a binary search tree (sorted by $y$-coordinate)




| | |
|---|---|
| **Space** | $O(n(\log n)^{d-1})$ |
| **Construction time** | $O(n(\log n)^{d-1})$ |
| **Range query time** | $O(s+(\log n)^d)$ |

* $d$ is constant

# Range Tree: Range Search.

*BST::RangeSearch*($r \leftarrow root$, $k_1$, $k_2$)
$r$: root of a binary search tree, $k_1, k_2$: search keys
Returns keys in subtree at $r$ that are in range $[k_1, k_2]$

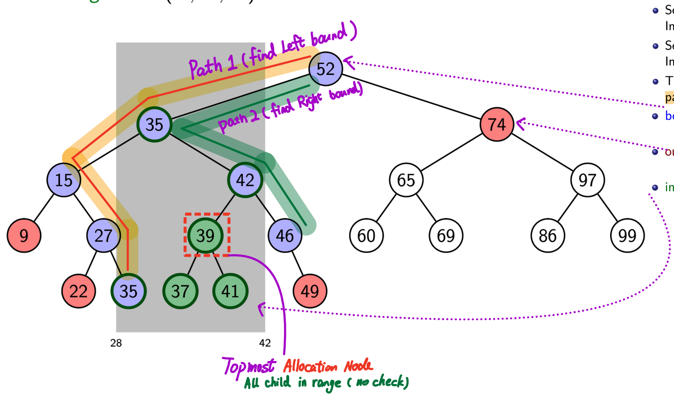
```
if r = NIL then return
if k1 ≤ r.key ≤ k2 then
    L ← BST::RangeSearch(r.left, k1, k2)
    R ← BST::RangeSearch(r.right, k1, k2)
    return L ∪ r.{key} ∪ R
if r.key < k1 then
    return BST::RangeSearch(r.right, k1, k2)
if T.key > k2 then
    return BST::RangeSearch(r.left, k1, k2)
```

Keys are reported in in-order, i. e., in sorted order.

Note: If there are *duplicates*, then this finds all copies that are in range.

- Search for path $P_1$: $O(\log n)$
- Search for path $P_2$: $O(\log n)$
- $O(\log n)$ boundary nodes

*BST::RangeSearch*($T$, 28, 42)




- Search for left boundary $k_1$: this gives path $P_1$
  In case of equality, go *left* to ensure that we find all duplicates.
- Search for right boundary $k_2$: this gives path $P_2$
  In case of equality, go *right* to ensure that we find all duplicates.
- This partitions $T$ into three groups: outside, on, or between the paths.
- boundary nodes: nodes in $P_1$ or $P_2$
  - For each boundary node, test whether it is in the range.
- outside nodes: nodes that are left of $P_1$ or right of $P_2$
  - These are *not* in the range, we stop the search at the topmost.
- inside nodes: nodes that are right of $P_1$ and left of $P_2$
  - We stop the search at the topmost (**allocation node**).
  - All descendants of an allocation node are *in* the range. For a 1d-range-search, report them.

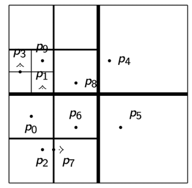
# Range Query Summary

| | | |
|---|---|---|
| (Range Tree) **Space** | $O(n(\log n)^{d-1})$ | kd-trees: $O(n)$ |
| (Range Tree) **Construction time** | $O(n(\log n)^{d-1})$ | kd-trees: $O(n\log n)$ |
| (Range Tree) **Range query time** | $O(s+(\log n)^d)$ | kd-trees: $O(s+n^{1-1/d})$ |

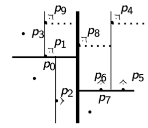
| | Range Query |
|---|---|
| Unsorted list hash table / array | $\Omega(n)$ |
| Sorted array | $O(\log n + s)$ |
| BST | $O(\text{height} + s)$ |

## Range query data structures summary

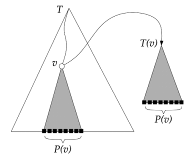
- Quadtrees
  - simple (also for dynamic set of points)
  - work well only if points evenly distributed
  - wastes space for higher dimensions



- kd-trees
  - linear space
  - query-time $O(\sqrt{n}+s)$
  - inserts/deletes destroy balance
  - care needed if not in general position



- range trees
  - query-time $O(\log^2 n + s)$
  - wastes some space
  - inserts/deletes destroy balance



**Convention:** Points on split lines belong to right/top side.

# Pattern Matching:

- Substring $T[i..j]$ $0 \le i \le j < n$: a string of length $j-i+1$ which consists of characters $T[i], \dots T[j]$ in order
- A prefix of $T$: 前缀
  a substring $T[0..i]$ of $T$ for some $0 \le i < n$
- A suffix of $T$: 后缀
  a substring $T[i..n-1]$ of $T$ for some $0 \le i \le n-1$

Pattern matching algorithms consist of guesses and checks:

- A **guess** is a position $i$ such that $P$ might start at $T[i]$. Valid guesses (initially) are $0 \le i \le n-m$.
- A **check** of a guess is a single position $j$ with $0 \le j < m$ where we compare $T[i+j]$ to $P[j]$. We must perform $m$ checks of a single correct guess, but may make (many) fewer checks of an incorrect guess.

## Brute-force Algorithm:

- Example: $T =$ abbbababbab, $P =$ abba

| a | b | b | b | a | b | a | b | b | a | b |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | b | a | | | | | | | |
| | a | | | | | | | | | |
| | | a | | | | | | | | |
| | | | a | | | | | | | |
| | | | | a | b | b | | | | |
| | | | | | a | | | | | |
| | | | | | | a | b | b | a | |

- What is the worst possible input?
  $P = a^{m-1}b$, $T = a^n$
- Worst case performance $\Theta((n-m+1)m)$

## Failure Array:

- The failure array $F$ of size $m$: $F[j]$ is defined as the length of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- $F[0] = 0$ P[1..j] is for prevent the case of the prefix and suffix = the string P
- If a mismatch occurs at $P[j] \ne T[i]$ we set $j \leftarrow F[j-1]$
- Consider $P =$ abacaba

| $j$ | $P[1..j]$ | $P$ | $F[j]$ |
|---|---|---|---|
| 0 | — | abacaba | 0 |
| 1 | b | abacaba | 0 |
| 2 | ba | abacaba | 1 |
| 3 | bac | abacaba | 0 |
| 4 | baca | abacaba | 1 |
| 5 | bacab | abacaba | 2 |
| 6 | bacaba | abacaba | 3 |

## KMP-Algorithm: 与后缀相同最长的前缀

- When a mismatch occurs, what is the most we can shift the pattern (reusing knowledge from previous matches)?

| $T =$ | a | b | c | d | c | a | b | c | ? | ? | ? |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | b | c | d | c | a | b | a | | | |
| | | | | | | a | b | c | d | c | a |

- KMP Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$

```
KMP(T, P)
T: String of length n (text), P: String of length m (pattern)
    F ← failureArray(P)
    i ← 0
    j ← 0
    while i < n do
        if T[i] = P[j] then
            if j = m − 1 then
                return i − j //match
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0 then
                j ← F[j − 1]
            else
                i ← i + 1     第一个就fail，直接看下一个
    return −1 // no match
```

$P =$ abacaba
$T =$ abaxyabacabbaababacaba

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | a | x | y | a | b | a | c | a | b | b |
| a | b | a | c | | | | | | | | |
| | | (a) | b | | | | | | | | |
| | | | a | | | | | | | | |
| | | | | a | | | | | | | |
| | | | | | a | b | a | c | a | b | a |
| | | | | | | | | | (a) | (b) | a |

failureArray

- At each iteration of the while loop, either
  1. $i$ increases by one, or
  2. the guess index $i-j$ increases by at least one ($F[j-1] < j$)
- There are no more than $2m$ iterations of the while loop
- Running time: $\Theta(m)$

KMP

- failureArray can be computed in $\Theta(m)$ time
- At each iteration of the while loop, either
  1. $i$ increases by one, or
  2. the guess index $i-j$ increases by at least one ($F[j-1] < j$)
- There are no more than $2n$ iterations of the while loop
- Running time: $\Theta(n)$

# Boyer-Moore Algorithm 坏字符还是好后缀?

## Last occurance Function: 该字符最后一次出现在 pattern 是在哪 ( index 从 0 开始)

- Preprocess the pattern $P$ and the alphabet $\Sigma$
- Build the last-occurrence function $L$ mapping $\Sigma$ to integers
- $L(c)$ is defined as
  - the largest index $i$ such that $P[i] = c$ or
  - $-1$ if no such index exists
- Example: $\Sigma = \{a, b, c, d\}$, $P = abacab$ (012345)

| $c$ | a | b | c | d |
|---|---|---|---|---|
| $L(c)$ | 4 | 5 | 3 | -1 |

- The last-occurrence function can be computed in time $O(m + |\Sigma|)$
- In practice, $L$ is stored in a size-$|\Sigma|$ array.

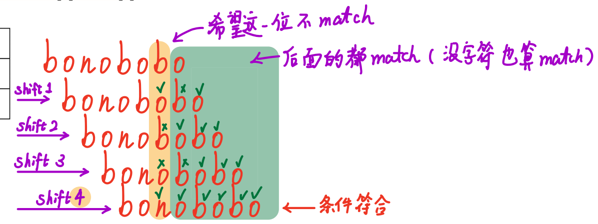
- Reverse-order searching: Compare $P$ with a subsequence of $T$ moving backwards
- Bad character jumps: When a mismatch occurs at $T[i] = c$
  - If $P$ contains $c$, we can shift $P$ to align the last occurrence of $c$ in $P$ with $T[i]$
  - Otherwise, we can shift $P$ to align $P[0]$ with $T[i+1]$
- Good suffix jumps: If we have already matched a suffix of $P$, then get a mismatch, we can shift $P$ forward to align with the previous occurence of that suffix (with a mismatch from the actual suffix). Similar to failure array in KMP.
- When a mismatch occurs, Boyer-Moore chooses whichever of bad character or good suffix shifts the pattern further to the right.

## Suffix skip Array: 非人话:

- Suffix skip array $S$ of size $m$: for $0 \le i < m$, $S[i]$ is the largest index $j$ such that $P[i+1..m-1] = P[j+1..j+m-1-i]$ and $P[j] \ne P[i]$.

人话:

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $P[i]$ | b | o | n | o | b | o | b | o |
| $S[i]$ | $-6$ | $-5$ | $-4$ | $-3$ | 2 | $-1$ | 2 | 6 |




当前找第6位, shift 4. 所以答案是 6-4=2.

```
boyer-moore(T,P)
L ← last occurrance array computed from P
S ← suffix skip array computed from P
i ← m − 1,     j ← m − 1
while i < n and j ≥ 0 do
    if T[i] = P[j] then
        i ← i − 1
        j ← j − 1
    else
        i ← i + m − 1 − min(L[T[i]], S[j])
        j ← m − 1
if j = −1 return i + 1
else return FAIL
```

(选小的) m=-2 后挪2位

Last-Occurrence Function:

pattern 的第 L[c] 位与当前 i 对齐

| $c$ | O | M | E | N |
|---|---|---|---|---|
| $L[c]$ | 6 | 7 | -1 | 5 |

Suffix-Skip Array:

O=6

pattern 从当前位置 i 前 S[j] 个位置开始

| $j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $P[j]$ | O | M | N | O | M | N | O | M |
| $S[j]$ | -3 | -2 | -1 | -3 | -2 | -1 | -2 | 6 |

n=5 (选小的)

M=6

pattern[5] 与当前 i 对齐

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| O | M | N | O | M | N | O | M |

Search:

不 match　　不 match

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| O | M | N | O | O | N | O | M | N | E | M | O | M | N | O | M | N | O | M |
| | | | | M | N | O | M | | | | | | | | | | | |
| | | | | | | | | | | | | | M | | | | | |
| | | | | | | | | | | N | O | M | [N] | O | M | | | |
| | | | | | | | | | | | (O) | (M) | (N) | (O) | (M) | N | O | M |

从后往前看

# Tries of Suffix / Suffix trees

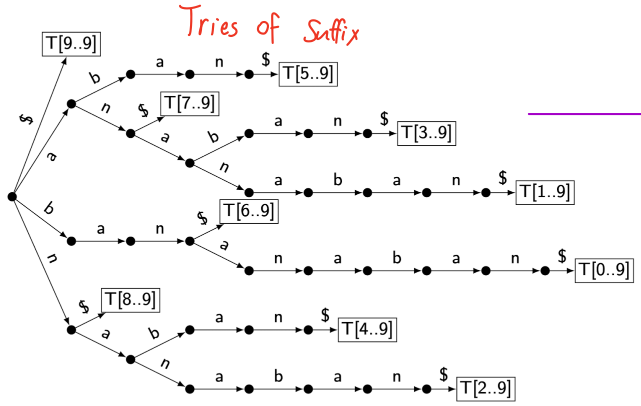
- What if we want to search for many patterns $P$ within the same fixed text $T$?
- To save space:
  - Use a compressed trie.
  - Store suffixes implicitly via indices into $T$.
- This is called a **suffix tree**.

- Text $T$ has $n$ characters and $n+1$ suffixes
- We can build the suffix tree by inserting each suffix of $T$ into a compressed trie. This takes time $\Theta(n^2)$. $(n+1)\cdot(\text{size of suffix}) \in \Theta(n^2)$
- There *is* a way to build a suffix tree of $T$ in $\Theta(n)$ time. This is quite complicated and beyond the scope of the course.

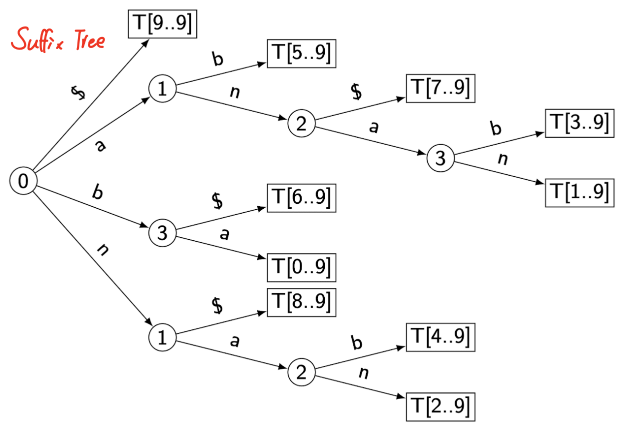
Store suffixes via indices: $T =$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| b | a | n | a | n | a | b | a | n | $ |

Tries of Suffix




Suffix Tree

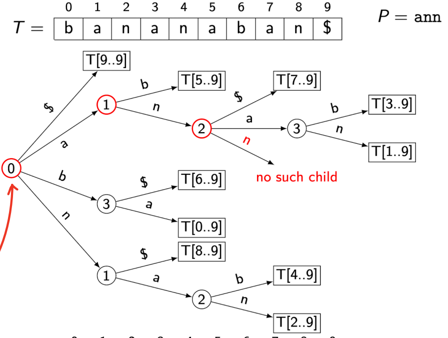


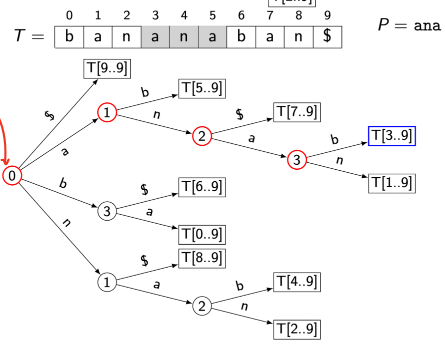
# String matching on Suffix trees

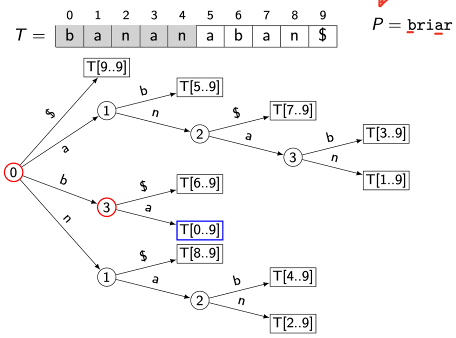
Assume we have a suffix tree of text $T$.

To search for pattern $P$ of length $m$:

- We assume that $P$ does not have the final \$.
- $P$ is the prefix of some suffix of $T$.
- In the *uncompressed* trie, searching for $P$ would be easy: $P$ exists in $T$ if and only if search for $P$ reaches a node in the trie.
- In the suffix tree, search for $P$ until one of the follow occurs:
  1. If search fails due to "*no such child*" then $P$ is not in $T$
  2. If we reach end of $P$, say at node $v$, then jump to leaf $\ell$ in subtree of $v$. (We presume that suffix trees stores such shortcuts.)
  3. Else we reach a leaf $\ell = v$ while characters of $P$ left.
- Either way, left index at $\ell$ gives the shift that we should check.
- This takes $O(|P|)$ time.




# Pattern match Summary: n: string 长度 m: pattern 长度

Failure Array Last Occurrence Fn

Suffix-skip Array

| | Brute-Force | KMP | Boyer-Moore | Suffix trees |
|---|---|---|---|---|
| Preprocessing: | – | $O(m)$ | $O(m + \|\Sigma\|)$ | $O(n^2)$ |
| Search time: | $O(nm)$ | $O(n)$ | $O(n)$ (often better) | $O(m)$ |
| Extra space: | – | $O(m)$ | $O(m + \|\Sigma\|)$ | $O(n)$ |

# Encoding

Compression Ratio: $\dfrac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}$

**Source text** The original data, string $S$ of characters from the **source alphabet** $\Sigma_S$

**Coded text** The encoded data, string $C$ of characters from the **coded alphabet** $\Sigma_C$

## Fix-length Code.

**Fixed-length code**: All codewords have the same length.

ASCII (American Standard Code for Information Interchange), 1963:

| char | null | start of heading | start of text | end of text | ... | 0 | 1 | ... | A | B | ... | ~ | delete |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| code | 0 | 1 | 2 | 3 | ... | 48 | 49 | ... | 65 | 66 | ... | 126 | 127 |

- 7 bits to encode 128 possible characters: "control codes", spaces, letters, digits, punctuation

  $A{\cdot}P{\cdot}P{\cdot}L{\cdot}E \to (65, 80, 80, 76, 69) \to$ 1000001 1010000 1010000 1001100 1000101

- Standard in *all* computers and often our source alphabet.
- Not well-suited for non-English text: ISO-8859 extends to 8 bits, handles most Western languages

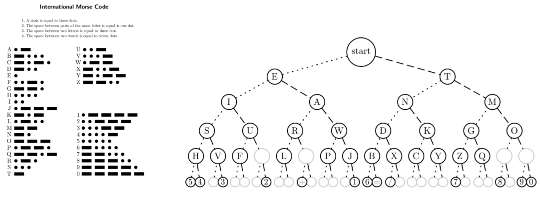
**Other (earlier) examples**: Caesar shift, Baudot code, Murray code

To decode a fixed-length code (say codewords have $k$ bits), we look up each $k$-bit pattern in a table.

## Variable-length code:

**Variable-length code**: Codewords may have different lengths.

**Example 1**: Morse code.



Pictures taken from http://apfelmus.nfshost.com/articles/fun-with-morse-code.html

**Example 2**: UTF-8 encoding of Unicode:

- Encodes any Unicode character (more than 107,000 characters) using 1-4 bytes

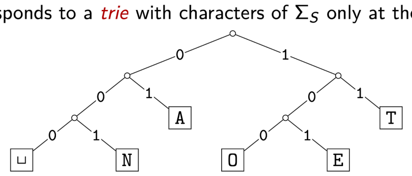
## Decoding:

The **decoding algorithm** must map $\Sigma_C^*$ to $\Sigma_S^*$.

- The code must be *uniquely decodable*.
  - This is false for Morse code as described!

    • ▬ ▬ ▬ • ▬ ▬ ▬ decodes to WATT and ANO and WJ.

    (Morse code uses 'end of character' pause to avoid ambiguity.)
- From now on only consider **prefix-free codes** $E$: no codeword is a prefix of another
- This corresponds to a *trie* with characters of $\Sigma_S$ only at the leaves.



- The codewords need no end-of-string symbol \$ if $E$ is prefix-free.

Any prefix-free code is uniquely decodable (why?)

```
PrefixFreeDecoding(T, C[0..n − 1])
T : the trie of a prefix-free code, C: text with characters in Σ_C
    initialize empty string S
    i ← 0
    while i < n
        r ← T.root
        while r is not a leaf
            if i = n return "invalid encoding"
            c ← child of r that is labelled with C[i]
            i ← i + 1
            r ← c
        S.append(character stored at r)
    return S
```

Run-time: $O(|C|)$.

## Encoding:

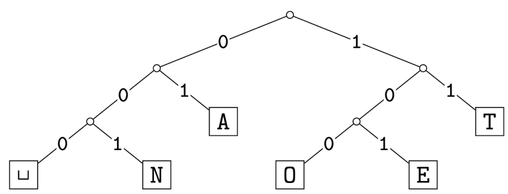
```
PrefixFreeEncodingFromTrie(T, S[0..n − 1])
T : the trie of a prefix-free code, S: text with characters in Σ_S
    L ← array of nodes in T indexed by Σ_S
    for all leaves ℓ in T
        L[character at ℓ] ← ℓ
    initialize empty string C
    for i = 0 to n − 1
        w ← empty string; v ← L[S[i]]
        while v is not the root
            w.prepend(character from v to its parent)
        // Now w is the encoding of S[i].
        C.append(w)
    return C
```

Run-time: $O(|T| + |C|) = O(|\Sigma_S| + |C|)$.

Code as table:

| $c \in \Sigma_S$ | ␣ | A | E | N | O | T |
|---|---|---|---|---|---|---|
| $E(c)$ | 000 | 01 | 101 | 001 | 100 | 11 |

Code as trie:



- Encode AN␣ANT → 010010000100111
- Decode 111000001010111 → TO␣EAT

## Huffman's Algorithm: 对频率高的text做优化

For a given source text $S$, how to determine the "best" trie that minimizes the length of $C$?

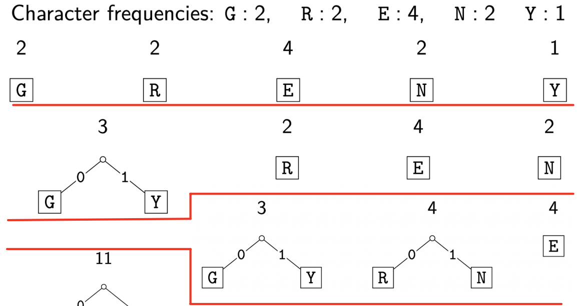
1. Determine frequency of each character $c \in \Sigma$ in $S$
2. For each $c \in \Sigma$, create "$\boxed{c}$" (height-0 trie holding $c$).
3. Our tries have a *weight*: sum of frequencies of all letters in trie. Initially, these are just the character frequencies.
4. Find the two tries with the minimum weight.
5. Merge these tries with new interior node; new weight is the sum. (Corresponds to adding one bit to the encoding of each character.)
6. Repeat last two steps until there is only one trie left

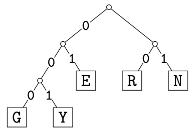
What data structure should we store the tries in to make this efficient?

A min-ordered heap! Step 4 is two *delete-min*s, Step 5 is *insert*

Example text: GREENENERGY, $\Sigma_S = \{G, R, E, N, Y\}$

Character frequencies: G : 2, R : 2, E : 4, N : 2 Y : 1





GREENENERGY → 000 10 01 01 11 01 11 01 10 000 001

Compression ratio: $\frac{25}{11 \cdot \log 5} \approx 97\%$

(These frequencies are not skewed enough to lead to good compression.)

```
Huffman-Encoding(S[0..n−1])
S: text over some alphabet Σ_S
    f ← array indexed by Σ_S, initially all-0            // frequencies
    for i = 0 to n − 1 do increase f[S[i]] by 1

    Q ← min-oriented priority queue that stores tries    // initialize PQ
    for all c ∈ Σ_S with f[c] > 0 do
        Q.insert(single-node trie for c with weight f[c])

    while Q.size > 1 do                                  // build decoding trie
        T1 ← Q.deleteMin(), f1 ←weight of T1
        T2 ← Q.deleteMin(), f2 ←weight of T2
        Q.insert(trie with T1, T2 as subtries and weight f1+f2)
    T ← Q.deleteMin

    C ← PrefixFreeEncodingFromTrie(T, S)
    return C and T
```

- Note: constructed trie is *not unique* (why?)
  So decoding trie must be transmitted along with the coded text $C$.
- This may make encoding bigger than source text!
- Encoding must pass through text twice (to compute frequencies and to encode)
- Encoding run-time: $O(|\Sigma_S| \log |\Sigma_S| + |C|)$
- Decoding run-time: $O(|C|)$
- The constructed trie is *optimal* in the sense that $C$ is shortest (among all prefix-free character-encodings with $\Sigma_C = \{0, 1\}$). We will not go through the proof.
- Many variations (give tie-breaking rules, estimate frequencies, adaptively change encoding, ....) Most frequency one should be shortest

# Run length encording

- Variable-length code
- Example of **multi-character encoding**: multiple source-text characters receive one code-word.
- The source alphabet and coded alphabet are both binary: $\{0, 1\}$.
- Decoding dictionary is uniquely defined and not explicitly stored.

**When to use**: if $S$ has long runs: $\underbrace{00000}\underbrace{111}\underbrace{0000}$

## Prefix Free for positive Integer: Elias Gamma Coding

Use **Elias gamma coding** to encode $k$:

- $\lfloor \log k \rfloor$ copies of 0, followed by
- binary representation of $k$ (always starts with 1)

| $k$ | $\lfloor \log k \rfloor$ | $k$ in binary | encoding |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
| 2 | 1 | 10 | 010 |
| 3 | 1 | 11 | 011 |
| 4 | 2 | 100 | 00100 |
| 5 | 2 | 101 | 00101 |
| 6 | 2 | 110 | 00110 |
| ⋮ | ⋮ | ⋮ | ⋮ |

## RLE Encoding

```
RLE-Encoding(S[0...n−1])
S: bitstring
    initialize output string C ← S[0]
    i ← 0                          // index of parsing S
    while i < n do
        k ← 1                      // length of run
        while (i + k < n and S[i + k] = S[i]) do k++
        i ← i + k

        // compute and append Elias gamma code
        K ← empty string
        while k > 1
            C.append(0)
            K.prepend(k mod 2)
            k ← ⌊k/2⌋
        K.prepend(1)               // K is binary encoding of k
        C.append(K)

    return C
```

## RLE Decoding

```
RLE-Decoding(C)
C: stream of bits
    initialize output string S
    b ← C.pop()        // bit-value for the current run
    repeat
        ℓ ← 0          // length of base-2 number −1
        while C.pop() = 0 do ℓ++

        k ← 1          // base-2 number converted
        for (j ← 1 to ℓ) do k ← k ∗ 2 + C.pop()

        for (j ← 1 to k) do S.append(b)
        b ← 1 − b
    until C has no more bits left
    return S
```

If $C.pop()$ is called when there are no bits left, then $C$ was not valid input.

Encoding:

$S = 11111110010000000000000000000011111111111$

$C = 1$ 第一个=开始是1 or 0

$S = 11111110010000000000000000000011111111111$

$k = 7$

$C = 100111$

$S = 11111110010000000000000000000011111111111$

$k = 2$

$C = 100111010$

$S = 11111110010000000000000000000011111111111$

$k = 1$

$C = 1001110101$

$S = 11111110010000000000000000000011111111111$

$k = 20$

$C = 1001110101000010100$

$S = 11111110010000000000000000000011111111111$

$k = 11$

$C = 10011101010000101000001011$

Compression ratio: $26/41 \approx 63\%$

Decoding:

$C = 00001101001001010$

$b = 0$

$C = 00001101001001010$

$b = 0$

$\ell = 3$

$C = 00001101001001010$

$b = 0$

$\ell = 3$

$k = 13$

$S = 0000000000000$

$C = 00001101001001010$

$b = 1$

$\ell = 2$

$k =$

$S = 0000000000000$

$C = 00001101001001010$

$b = 1$

$\ell = 2$

$k = 4$

$S = 00000000000001111$

$C = 00001101001001010$

$b = 1$

$\ell = 2$

$k = 4$

$S = 00000000000001111$

$C = 00001101001001010$

$b = 1$

$\ell = 1$

$k =$

$S = 000000000000011110$

$C = 00001101001001010$

$b = 1$

$\ell = 1$

$k = 2$

$S = 00000000000001111011$

- An all-0 string of length $n$ would be compressed to $2\lfloor \log n \rfloor + 2 \in o(n)$ bits.
- Usually, we are not that lucky:
  - No compression until run-length $k \geq 6$
  - *Expansion* when run-length $k = 2$ or $4$
- Used in some image formats (e.g. TIFF)
- Method can be adapted to larger alphabet sizes (but then the encoding of each run must also store the character)
- Method can be adapted to encode *only* runs of 0 (we will need this soon)

# Lempel-Ziv-Welch Compression / LZW compression

**Ingredient 1** for Lempel-Ziv-Welch compression: take advantage of such substrings *without* needing to know beforehand what they are.

- English text:
  Most frequent digraphs: `TH, ER, ON, AN, RE, HE, IN, ED, ND, HA`
  Most frequent trigraphs: `THE, AND, THA, ENT, ION, TIO, FOR, NDE`
- HTML: "`<a href`", "`<img src`", "`<br>`"
- Video: repeated background between frames, shifted sub-image

**Ingredient 2** for LZW: *adaptive encoding*:

- There is a fixed initial dictionary $D_0$. (Usually ASCII.)
- For $i \geq 0$, $D_i$ is used to determine the $i$th output character
- After writing the $i$th character to output, both encoder and decoder update $D_i$ to $D_{i+1}$

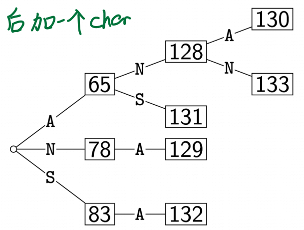
## Overview:

- Start with dictionary $D_0$ for $|\Sigma_S|$.
  Usually $\Sigma_S = ASCII$, then this uses codenumbers $0, \ldots, 127$.
- Every step adds to dictionary a multi-character string, using codenumbers $128, 129, \ldots$.
- Encoding:
  - Store current dictionary $D_i$ as a trie.
  - Parse trie to find longest prefix $w$ already in $D_i$.
    So all of $w$ can be encoded with one number.
  - Add to dictionary the *substring that would have been useful*:
    add $wK$ where $K$ is the character that follows $w$ in $S$.
  - This creates one child in trie at the leaf where we stopped.
- Output is a list of numbers. This is usually converted to bit-string with fixed-width encoding using 12 bits.
  - This limits the codenumbers to 4096.

| Text: | A | N | A N | A | S | A N | N A |
|---|---|---|---|---|---|---|---|
| | 65 | 78 | 128 | 65 | 83 | 128 | 129 |

每次只往已知串后加一个char

Dictionary:



Final output: $\underbrace{000001000001}_{65}\underbrace{000001001110}_{78}\underbrace{000010000000}_{128}\underbrace{000001000001}_{65}\underbrace{000001010011}_{83}\underbrace{000010000000}_{128}\underbrace{000010000001}_{129}$

```
LZW-encode(S)
S : stream of characters
1.  Initialize dictionary D with ASCII in a trie
2.  idx ← 128
3.  while there is input in S do
4.      v ← root of trie D
5.      K ← S.peek()
6.      while (v has a child c labelled K)
7.          v ← c; S.pop()
8.          if there is no more input in S break    (goto 10)
9.          K ← S.peek()
10.     output codenumber stored at v
11.     if there is more input in S
12.         create child of v labelled K with codenumber idx
13.         idx++
```

## Decoding:

- Same idea: build dictionary while reading string.
- Dictionary maps numbers to strings.
  To save space, store string as code of prefix + one character.
- Example: 67 65 78 32 66 129 133

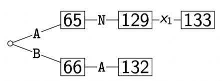
$D =$

| Code # | String |
|---|---|
| ... | |
| 32 | ␣ |
| ... | |
| ... | |
| 65 | A |
| 66 | B |
| 67 | C |
| ... | |
| 78 | N |
| ... | |
| 83 | S |
| ... | |

| input | decodes to | Code # | String (human) | String (computer) |
|---|---|---|---|---|
| 67 | C | | | |
| 65 | A | 128 | CA | 67, A |
| 78 | N | 129 | AN | 65, N |
| 32 | ␣ | 130 | N␣ | 78, ␣ |
| 66 | B | 131 | ␣B | 32, B |
| 129 | AN | 132 | BA | 66, A |
| 133 | ??? | 133 | | |

| Text: | C | A | N | ␣ | B | A N | $x_1$ | $x_2$ | ... |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | A | N | $x_1$ |
| | 67 | 65 | 78 | 33 | 66 | 129 | 133 | | |

Dictionary (parts omitted):



- We know: 133 encodes $\mathtt{AN}x_1$ (for unknown $x_1$)
- We know: Next step uses $133 = \mathtt{AN}x_1$
- So $x_1 =$A and 133 encodes ANA

| input | decodes to | Code # | String (human) | String (computer) |
|---|---|---|---|---|
| 67 | C | | | |
| 65 | A | 128 | CA | 67, A |
| 78 | N | 129 | AN | 65, N |
| 32 | ␣ | 130 | N␣ | 78, ␣ |
| 66 | B | 131 | ␣B | 32, B |
| 129 | AN | 132 | BA | 66, A |
| 133 | ANA | 133 | ANA | 129, A |
| 83 | S | 134 | ANAS | 133, S |

Generally: If code number is about to be added to $D$, then it encodes

"previous string + first character of previous string"

```
LZW-decode(C)
C: stream of integers
D ← dictionary that maps {0, ..., 127} to ASCII
idx ← 128
S ← empty string

code ← C.pop(); s ← D(code); S.append(s)
while there are more codes in C do
    s_prev ← s; code ← C.pop()

    if code < idx
        s ← D(code)
    else if code = idx  // special situation!
        s ← s_prev + s_prev[0]
    else FAIL           // Encoding was invalid

    S.append(s)
    D.insert(idx, s_prev + s[0])
    idx++
return S
```

## LZW Summary

- Encoding: $O(|S|)$ time, uses a trie of encoded substrings to store the dictionary
- Decoding: $O(|S|)$ time, uses an array indexed by code numbers to store the dictionary.
- Encoding and decoding need to go through the string only *once* and do not need to see the whole string
  $\Rightarrow$ can do compression while streaming the text
- Compresses quite well ($\approx 45\%$ on English text).

bzip2

| | | |
|---|---|---|
| text $T_0$ | | |
| ↓ | Burrows-Wheeler transform | If $T_0$ has repeated substrings, then $T_1$ has long runs of characters. |
| text $T_1$ | | |
| ↓ | Move-to-front transform | If $T_1$ has long runs of characters, then $T_2$ has long runs of zeros and skewed frequencies. |
| text $T_2$ | | |
| ↓ | Modified RLE | If $T_2$ has long runs of zeroes, then $T_3$ is shorter. Skewed frequencies remain. |
| text $T_3$ | | |
| ↓ | Huffman encoding | Compresses well since frequencies are skewed. |
| text $T_4$ | | |

# Move-to-Front Transform 把读了的往前放

Recall the MTF heuristic for self-organizing search:

- Dictionary $L$ is stored as an unsorted array or linked list
- After an element is accessed, move it to the front of the dictionary

How can we use this idea for transforming a text with repeat characters?

- Encode each character of source text $S$ by its index in $L$.
- After each encoding, update $L$ with Move-To-Front heuristic.
- **Example**: $S =$ GOOD becomes $C = 1, 2, 0, 2$

```
0 1 2      G      0 1 2      O      0 1 2      O      0 1 2      D      0 1 2
[D|G|O]   --->   [G|D|O]   --->   [O|G|D]   --->   [O|G|D]   --->   [D|O|G]
           1                2                0                2
```

**Observe:** A character in $S$ repeats $k$ times $\Leftrightarrow$ $C$ has run of $k-1$ zeroes

**Observe:** $C$ contains lots of small numbers and few big ones.

$C$ has the same length as $S$, but better properties.

## Move-to-Front Encoding/Decoding

```
MTF-encode(S)
1.  L ← array with Σ_S in some pre-agreed, fixed order (usually ASCII)
2.  while S has more characters do
3.      c ← next character of S
4.      output index i such that L[i] = c
5.      for j = i − 1 down to 0
6.          swap L[j] and L[j + 1]
```

Decoding works in *exactly* the same way:

```
MTF-decode(C)
1.  L ← array with Σ_S in some pre-agreed, fixed order (usually ASCII)
2.  while C has more characters do
3.      i ← next integer from C
4.      output L[i]
5.      for j = i − 1 down to 0
6.          swap L[j] and L[j + 1]
```

# Burrow-Wheeler Transform

**Idea:**

- *Permute* the source text $S$: the coded text $C$ has the exact same letters (and the same length), but in a different order.
- **Goal:** If $S$ has repeated substrings, then $C$ should have long runs of characters.
- We need to choose the permutation carefully, so that we can *decode* correctly.

**Details:**

- Assume that the source text $S$ ends with end-of-word character \$ that occurs nowhere else in $S$.
- A **cyclic shift** of $S$ is the concatenation of $S[i+1..n-1]$ and $S[0..i]$, for $0 \le i < n$.
- The encoded text $C$ consists of the last characters of the cyclic shifts of $S$ after sorting them.

## Overview

**Encoding cost**: $O(n^2)$ (using MSD radix sort) and often better

Encoding is theoretically possible in $O(n)$ time:

- Sorting cyclic shifts of $S$ is equivalent to sorting the suffixes of $S \cdot S$ that have length $> n$
- This can be done by traversing the suffix tree of $S \cdot S$

**Decoding cost**: $O(n)$ (faster than encoding)

Encoding and decoding both use $O(n)$ space.

They need *all* of the text (no streaming possible). BWT is a **block compression method**.

BWT tends to be slower than other methods, but (combined with MTF, modified RLE and Huffman) gives better compression.

## Encoding

$S =$ alf␣eats␣alfalfa\$

| 1 Write all cyclic shifts | 2 Sort cyclic shifts | 3 Extract last characters from sorted shifts |
|---|---|---|
| alf␣eats␣alfalfa\$ | \$alf␣eats␣alfalfa | \$alf␣eats␣alfalfa |
| lf␣eats␣alfalfa\$a | ␣alfalfa\$alf␣eats | ␣alfalfa\$alf␣eats |
| f␣eats␣alfalfa\$al | ␣eats␣alfalfa\$alf | ␣eats␣alfalfa\$alf |
| ␣eats␣alfalfa\$alf | a\$alf␣eats␣alfalf | a\$alf␣eats␣alfalf |
| eats␣alfalfa\$alf␣ | alf␣eats␣alfalfa\$ | alf␣eats␣alfalfa\$ |
| ats␣alfalfa\$alf␣e | alfa\$alf␣eats␣alf | alfa\$alf␣eats␣alf |
| ts␣alfalfa\$alf␣ea | alfalfa\$alf␣eats␣ | alfalfa\$alf␣eats␣ |
| s␣alfalfa\$alf␣eat | ats␣alfalfa\$alf␣e | ats␣alfalfa\$alf␣e |
| ␣alfalfa\$alf␣eats | eats␣alfalfa\$alf␣ | eats␣alfalfa\$alf␣ |
| alfalfa\$alf␣eats␣ | f␣eats␣alfalfa\$al | f␣eats␣alfalfa\$al |
| lfalfa\$alf␣eats␣a | fa\$alf␣eats␣alfal | fa\$alf␣eats␣alfal |
| falfa\$alf␣eats␣al | falfa\$alf␣eats␣al | falfa\$alf␣eats␣al |
| alfa\$alf␣eats␣alf | lf␣eats␣alfalfa\$a | lf␣eats␣alfalfa\$a |
| lfa\$alf␣eats␣alfa | lfa\$alf␣eats␣alfa | lfa\$alf␣eats␣alfa |
| fa\$alf␣eats␣alfal | lfalfa\$alf␣eats␣a | lfalfa\$alf␣eats␣a |
| a\$alf␣eats␣alfalf | s␣alfalfa\$alf␣eat | s␣alfalfa\$alf␣eat |
| \$alf␣eats␣alfalfa | ts␣alfalfa\$alf␣ea | ts␣alfalfa\$alf␣ea |

$C =$ asff\$f␣e␣lllaaata

## Decoding

**Idea**: Given $C$, we can reconstruct the *first* and *last column* of the array of cyclic shifts by sorting.

C = ard\$rcaaaabb

| 1 Last column: $C$ | 2 First column: $C$ sorted | 3 Disambiguate by row-index | 4 Starting from \$, recover $S$ |
|---|---|---|---|
| ..........a | \$.........a | \$,3.........a,0 | \$,3.........a,0 |
| ..........r | a.........r | a,0.........r,1 | a,0.........r,1 |
| ..........d | a.........d | a,6.........d,2 | a,6.........d,2 |
| ..........\$ | a.........\$ | a,7.........\$,3 | a,7.........\$,3 ← start |
| ..........r | a.........r | a,8.........r,4 | a,8.........r,4 |
| ..........c | a.........c | a,9.........c,5 | a,9.........c,5 |
| ..........a | b.........a | b,10.........a,6 | b,10.........a,6 |
| ..........a | b.........a | b,11.........a,7 | b,11.........a,7 |
| ..........a | c.........a | c,5.........a,8 | c,5.........a,8 |
| ..........a | d.........a | d,2.........a,9 | d,2.........a,9 |
| ..........b | r.........b | r,1.........b,10 | r,1.........b,10 |
| ..........b | r.........b | r,4.........b,11 | r,4.........b,11 |

S = abra ....

左边按顺序放入S

# Compression Summary

| **Huffman** | **Run-length encoding** | **Lempel-Ziv-Welch** | **bzip2** (uses Burrows-Wheeler) |
|---|---|---|---|
| variable-length | variable-length | fixed-length | multi-step |
| single-character | multi-character | multi-character | multi-step |
| 2-pass, must send dictionary | 1-pass | 1-pass | not streamable |
| 60% compression on English text | bad on text | 45% compression on English text | 70% compression on English text |
| optimal 01-prefix-code | good on long runs (e.g., pictures) | good on English text | better on English text |
| requires uneven frequencies | requires runs | requires repeated substrings | requires repeated substrings |
| rarely used directly | rarely used directly | frequently used | used but slow |
| part of pkzip, JPEG, MP3 | fax machines, old picture-formats | GIF, some variants of PDF, compress | bzip2 and variants |